

- 在这里，众多知名企业面试官将为你揭开神秘的求职面纱
- 在这里，多位求职达人将现身说法为你解开求职谜团
- 在这里，各类企业的招聘细节都会为你详尽展示

程序员面试笔试宝典

何昊 叶向阳 窦浩 编著



本书覆盖了历年来各大IT名企95%以上的面试笔试题，当你细细品读完本书的知识后，各类企业的offer将任由你挑选。本书将带你走进神奇的求职之旅。



机械工业出版社
CHINA MACHINE PRESS



程序员面试笔试宝典

何 昊 叶向阳 窦 浩 编著



机械工业出版社

本书针对当前各大 IT 企业面试笔试中常见的问题以及注意事项,进行了深层次的分析。本书除了对传统的计算机相关知识(C/C++、数据结构与算法、操作系统、计算机网络与通信、软件工程、数据库、智力题、英语面试等)进行介绍外,还根据当前计算机技术的发展潮流,对面试笔试中常见的海量数据处理进行了详细的分析。同时,为了更具说服力,本书特邀多位 IT 名企面试官现身说法,对面试过程中求职者存在的问题进行了深度剖析,同时本书引入了一批来自于名牌高校、就职于明星企业的职场达人的真实求职案例,通过他们的求职经验与教训,抛砖引玉,将整个求职过程生动形象地展示在读者面前,进而对求职者起到一定的指引作用。本书也对各种类型的 IT 企业的招聘环节进行了庖丁解牛式的分析,帮助求职者能够更加有针对性地进行求职准备。

本书是一本计算机相关专业毕业生面试笔试的求职用书,同时也适合期望在计算机软硬件行业大显身手的计算机爱好者阅读。

图书在版编目(CIP)数据

程序员面试笔试宝典 / 何昊, 叶向阳, 窦浩编著. —北京: 机械工业出版社, 2012.10

ISBN 978-7-111-39879-0

I. ①程… II. ①何… ②叶… ③窦… III. ①程序设计—工程技术人员—职业选择—基本知识 IV. ①C913.2

中国版本图书馆 CIP 数据核字(2012)第 228884 号

机械工业出版社(北京市百万庄大街 22 号 邮政编码 100037)

责任编辑: 时 静 范成欣

责任印制: 张 楠

北京双青印刷厂印刷

2012 年 10 月第 1 版·第 1 次印刷

184mm×260mm·26.5 印张·654 千字

0 001—3 500 册

标准书号: ISBN 978-7-111-39879-0

定价: 59.80 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

电话服务

网络服务

社服务中心: (010) 88361066

教材网: <http://www.cmpedu.com>

销售一部: (010) 68326294

机工官网: <http://www.cmpbook.com>

销售二部: (010) 88379649

机工官博: <http://weibo.com/cmp1952>

读者购书热线: (010) 88379203

封面无防伪标均为盗版

前言

21 世纪的前 10 年是 IT 技术迅速发展的 10 年，嵌入式技术、互联网技术等蓬勃发展，程序员（尤其是高级程序员）可观的薪水以及巨大的发展潜力使得越来越多的人选择了程序员作为未来的职业。同时，由于计算机技术自身博大精深，涉及的知识面很广，企业在给程序员开出高薪的同时，对他们的专业素养也提出了非常高的要求。每年 9 月份开始，各大 IT 企业开始招贤纳士，无数具有计算机专业背景的学生都希望在招聘季能够进入自己渴望的企业，并且领取一份可观的薪水。但是这些企业的招聘人数毕竟是有限的，像 Microsoft 这样的行业巨头，每年在中国的招聘计划可能只有区区几十人，如何能够鹤立鸡群，从众多求职者中脱颖而出是每一个 IT 求职者所渴望的。

通过调查发现，IT 业在对求职者进行面试笔试时，都会特别看重求职者对基础知识的掌握程度，重点考核他们的计算机基本知识，包括 C/C++、数据结构与算法、操作系统、计算机网络、大规模数据处理等。毕竟这些具体技术问题是具有客观答案的，而且计算机技术之间往往都是相通的，这些基础题型几乎是固定的，不管求职者之前的技术水平如何，只要能够在找工作前将这些知识理解，做到融会贯通，以不变应万变，必将能在求职中找到一份理想的工作。

当前市面上尽管有为数不少的介绍程序员面试笔试的书籍，可是却无法完全满足程序员求职者的需求，主要表现在以下几个方面：知识面太浅显，浮于表面，无法满足大型互联网公司、嵌入式软件公司等对人才的需求；未能引入一些职场达人的求职经历，无法给予求职者一些实际的案例、场景；知识点的侧重不是很清晰，对当前程序员面试笔试的重心把握不够明确；未能针对不同类型企业进行有针对性的分析，无法给予求职者有用的求职建议。本书编者以自身的实际经验以及一些职场达人的求职经验为蓝本，将求职过程中遇到的技术问题一一整理，试图将面试笔试这一复杂紧张的过程简单化，让每一个计算机相关专业的求职者都能通过本书找到一份满意的工作。

本书不同于同类书籍，其特点主要表现在以下几个方面：

1. 面试官箴言。本书邀请一些在知名企业从事过面试工作的软件工程师、系统分析师以及开发经理，请他们结合自己的亲身经历，详细地分析了面试官心理，并且对面试过程中求职者应该注意的事项以及求职者如何准备才能获得面试官的青睐进行了深入的剖析，对求职者求职具有非常好的标杆作用。

2. “面经”真实、有代表性。本书将当前中国实力最强劲的计算机类大学或研究所的求职达人（他们分别来自中国科学院计算技术研究所、西安电子科技大学、浙江大学、中山大学、电子科技大学等名牌大学）的求职经历引入，他们的就业前景（包括人民搜索、腾讯搜索、网易游戏、睿初科技、支付宝、华为等）代表了优秀的就业目标，具有非常高的代表性与就业参考价值。

3. 知识点广、深。相比较其他同类书籍，本书知识面更广，知识点更深，更加适合当前

IT 企业对优秀人才的需求。随着计算机教育的遍地开花，计算机人才的层次却参差不齐，一方面广大 IT 企业遭遇用人荒；而另一方面，广大计算机培育机构无法培养出符合市场需求的高素质计算机人才，

4. 更有针对性。本书将各大类型企业的招聘流程、面试笔试注意事项、真题一一展现在读者面前，同时针对这些原始资料进行深度剖析，使读者能够更有针对性地准备不同性质的企业。同时，本书还分析了求职者在选择 offer 的时候，如何结合自己的真实情况来选择一个适合自己发展的企业，防止求职者不经过深思熟虑，盲目地选择企业。

5. 有所倚重。本书的侧重点放在各大 IT 企业更关心的问题，尤其是最常见的语言知识、算法与数据结构、海量数据处理等，对不同类型的企业，更有说服力。

本书的出版得到了机械工业出版社时静的大力支持。本书由何昊、叶向阳、窦浩共同编写，黎建文、王幅钹、周明媛、马轩等提供了部分内容的原始资料；西安电子科技大学软件安全与编译器架构实验室的硕士研究生厉孙德、张振鹏、邢芸茜、黄珊珊、池浩、柴艳瑞、刘琛、苗永强，计算机学院的薛鹏、刘倩、杨静、裴帅帅，电子工程学院的柴睿都从不同方面做了一些工作；董西成、邵帅、王震、伍文明、李超、代俊、曹润涛、郭晶晶、阎贝、林方超、廖兰新、李志强、胥济国、赵威、豆云、许胜之、王蕾、衡量、何芳等好朋友为本书的内容也提供了非常宝贵的材料；特别感谢褚艳利、丁志浩、徐磊、卢山等师长，他们在百忙中抽出时间，以自己的亲身经历对程序员面试中求职者存在的问题，站在面试官的角度提供了很多非常有意义的意见与建议；西安电子科技大学软件学院的刘坚教授、张立勇副教授、王献青副教授、霍秋艳副教授对本人的成长给予了极大的关心与照顾；何四为律师为本书提供了一些有关版权的援助，在此向他们致以衷心的感谢。

在本书的编写过程中，还得到了武方方主任、张向虎主任、冯佩燕处长、鲁昊鹏高工、张剑高工、张玉博高工、谭宏光高工、谢卫高工、赵亮高工、徐晓乐高工、马红日高工，同事薛波、杜林、郭肖晓、王晓、梁冰、苏春宇、罗星原、陈皓、李刚、刘雷、渠慎建、单晓明、张博、尹茂旸、回永利、姜敏、张翔、梁昊、梁勇等的大力支持，在此一并致以最衷心的感谢。

一本高质量的图书从构思到出版，期间耗费的巨大精力是常人难以想象的，令人欣慰的是，在此过程中得到了诸多朋友、同学、领导、同事、家人、亲戚等的鼓励与激励，没有他们，我们很难在繁重的学习、工作之余完成此书的编写工作。尽管编者尽了最大的努力来创作本书，但是由于学识浅薄、见闻不广，不足之处在所难免，希望得到谅解与指正。

编 者

目 录

前言

上篇 面试笔试经验技巧篇

| | | |
|-------|-------------------------|----|
| 第 1 章 | 面试官箴言 | 2 |
| 1.1 | 有道无术，术可求；有术无道，止于术 | 2 |
| 1.2 | 求精不求全 | 3 |
| 1.3 | 脚踏实地，培养多种技能 | 4 |
| 1.4 | 保持空杯心态 | 6 |
| 1.5 | 职场是能者的舞台 | 7 |
| 1.6 | 学会“纸上谈兵” | 8 |
| 1.7 | 小结 | 8 |
| 第 2 章 | 面试心得交流 | 9 |
| 2.1 | 心态决定一切 | 9 |
| 2.2 | 假话全不说，真话不全说 | 10 |
| 2.3 | 走自己的路，让别人去说吧 | 12 |
| 2.4 | 夯实基础谋出路 | 14 |
| 2.5 | 书中自有编程法 | 15 |
| 2.6 | 笔试成绩好，不会被鄙视 | 17 |
| 2.7 | 不要一厢情愿做公司的备胎 | 18 |
| 2.8 | 小结 | 19 |
| 第 3 章 | 企业面试笔试攻略 | 20 |
| 3.1 | 互联网企业 | 20 |
| 3.2 | 网络设备提供商 | 25 |
| 3.3 | 外企 | 29 |
| 3.4 | 国企 | 32 |
| 3.5 | 研究所 | 35 |
| 3.6 | 创业型企业 | 37 |
| 3.7 | 如何抉择 | 41 |
| 第 4 章 | 面试笔试技巧 | 42 |
| 4.1 | 不打无准备之仗 | 42 |
| 4.1.1 | 如何获取求职信息 | 42 |
| 4.1.2 | 如何制作一份受用人单位青睐的简历 | 43 |
| 4.1.3 | 如何高效地网申简历 | 47 |
| 4.1.4 | 面试考查什么内容 | 48 |
| 4.1.5 | 霸王面合适吗 | 50 |
| 4.1.6 | 非技术类笔试如何应答 | 50 |
| 4.1.7 | 什么是职场暗语 | 51 |
| 4.1.8 | 如何克服面试中的紧张情绪 | 54 |
| 4.1.9 | 面试礼仪有哪些 | 55 |

| | | |
|--------|-----------------------|----|
| 4.1.10 | 面试需要准备什么内容 | 56 |
| 4.1.11 | 女生适合做程序员吗 | 57 |
| 4.1.12 | 程序员是吃青春饭的吗 | 58 |
| 4.1.13 | 为什么会被企业拒绝 | 58 |
| 4.1.14 | 如何准备集体面试 | 59 |
| 4.1.15 | 如何准备电话面试 | 61 |
| 4.2 | 从容应对 | 62 |
| 4.2.1 | 如何进行自我介绍 | 63 |
| 4.2.2 | 你对我们公司有什么了解 | 64 |
| 4.2.3 | 如何应对自己不会回答的问题 | 65 |
| 4.2.4 | 如何应对面试官的“激将法”语言 | 65 |
| 4.2.5 | 如何处理与面试官持不同观点的问题 | 66 |
| 4.2.6 | 如果你在这次面试中没有被录用,你会怎么办 | 66 |
| 4.2.7 | 如果你被我们录取了,接下来你将如何开展工作 | 66 |
| 4.2.8 | 你怎么理解你应聘的职位 | 67 |
| 4.2.9 | 你有哪些缺点 | 67 |
| 4.2.10 | 你有哪些优点 | 68 |
| 4.2.11 | 你没有工作经验,如何能够胜任这个岗位 | 69 |
| 4.2.12 | 你的好朋友是如何评价你的 | 69 |
| 4.2.13 | 你与上司意见不一致时,该怎么办 | 70 |
| 4.2.14 | 你能说说你的家庭吗 | 71 |
| 4.2.15 | 你认为自己最适合做什么 | 72 |
| 4.2.16 | 你如何看待公司的加班现象 | 72 |
| 4.2.17 | 你的业余爱好是什么 | 73 |
| 4.2.18 | 你和别人发生过争执吗?你怎样解决 | 74 |
| 4.2.19 | 你如何面对压力 | 74 |
| 4.2.20 | 你为什么离开了原来的单位 | 75 |
| 4.2.21 | 你为什么更倾向于我们公司 | 75 |
| 4.2.22 | 你觉得我们为什么要录用你 | 76 |
| 4.2.23 | 你的职业规划是什么 | 76 |
| 4.2.24 | 你对薪资有什么要求 | 77 |
| 4.2.25 | 你有什么需要问我的问题吗 | 77 |
| 4.3 | 签约这点事 | 78 |
| 4.3.1 | 风萧萧兮易水寒, offer 多了怎么办 | 78 |
| 4.3.2 | 签约、违约需要注意哪些事项 | 78 |
| 4.4 | 小结 | 81 |
| 第5章 | 英文面试攻略 | 82 |
| 5.1 | 注意事项 | 82 |
| 5.2 | 英文自我介绍 | 83 |
| 5.3 | 常见的英文面试问题 | 85 |
| 5.4 | 常见计算机专业词汇 | 94 |
| 5.4.1 | 计算机专业相关课程 | 94 |
| 5.4.2 | 操作系统相关术语 | 95 |

| | | |
|--------------|--------------------|------------|
| 5.4.3 | 算法相关术语 | 96 |
| 5.4.4 | 数据结构相关术语 | 97 |
| 5.4.5 | 计算机网络相关术语 | 100 |
| 第 6 章 | 智力题攻略 | 102 |
| 6.1 | 推理类 | 102 |
| 6.2 | 博弈类 | 107 |
| 6.3 | 计算类 | 109 |
| 6.4 | 作图类 | 111 |
| 6.5 | 倒水类 | 112 |
| 6.6 | 称重类 | 113 |
| 6.7 | 最优化类 | 114 |
| 6.8 | IT 思想类 | 115 |
| 6.9 | 过桥类 | 118 |
| 6.10 | 概率类 | 119 |

下篇 面试笔试技术攻克篇

| | | |
|--------------|--|------------|
| 第 7 章 | 程序设计基础 | 122 |
| 7.1 | C/C++关键字 | 122 |
| 7.1.1 | static (静态) 变量有什么作用 | 122 |
| 7.1.2 | const 有哪些作用 | 124 |
| 7.1.3 | switch 语句中的 case 结尾是否必须添加 break 语句? 为什么 | 127 |
| 7.1.4 | volatile 在程序设计中有什么作用 | 128 |
| 7.1.5 | 断言 ASSERT() 是什么 | 129 |
| 7.1.6 | 枚举变量的值如何计算 | 130 |
| 7.1.7 | char str1[] = "abc"; char str2[] = "abc"; str1 与 str2 不相等, 为什么 | 130 |
| 7.1.8 | 为什么有时候 main() 函数会带参数? 参数 argc 与 argv 的含义是什么 | 131 |
| 7.1.9 | C++ 里面是不是所有的动作都是 main() 函数引起的 | 132 |
| 7.1.10 | *p++ 与 (*p)++ 等价吗? 为什么 | 132 |
| 7.1.11 | 前置运算与后置运算有什么区别 | 132 |
| 7.1.12 | a 是变量, 执行(a++) += a 语句是否合法 | 133 |
| 7.1.13 | 如何进行 float、bool、int、指针变量与“零值”的比较 | 134 |
| 7.1.14 | new/delete 与 malloc/free 的区别是什么 | 135 |
| 7.1.15 | 什么时候需要将引用作为返回值 | 137 |
| 7.1.16 | 变量名为 618Software 是否合法 | 137 |
| 7.1.17 | C 语言中, 整型变量 x 小于 0, 是否可知 x×2 也小于 0 | 138 |
| 7.1.18 | exit(status) 是否跟从 main() 函数返回的 status 等价 | 138 |
| 7.1.19 | 已知 String 类定义, 如何实现其函数体 | 138 |
| 7.1.20 | 在 C++ 中如何实现模板函数的外部调用 | 140 |
| 7.1.21 | 在 C++ 中, 关键字 explicit 有什么作用 | 140 |
| 7.1.22 | C++ 中异常的处理方法以及使用了哪些关键字 | 141 |
| 7.1.23 | 如何定义和实现一个类的成员函数为回调函数 | 141 |
| 7.2 | 内存分配 | 142 |
| 7.2.1 | 内存分配的形式有哪些 | 142 |

| | | |
|--------|---|-----|
| 7.2.2 | 什么是内存泄露 | 143 |
| 7.2.3 | 栈空间的最大值是多少 | 144 |
| 7.2.4 | 什么是缓冲区溢出 | 144 |
| 7.3 | sizeof | 146 |
| 7.3.1 | sizeof 是关键字吗 | 146 |
| 7.3.2 | strlen("\0")=? sizeof("\0")=? | 146 |
| 7.3.3 | 对于结构体而言,为什么 sizeof 返回的值一般大于期望值 | 148 |
| 7.3.4 | 指针进行强制类型转换后与地址进行加法运算,结果是什么 | 149 |
| 7.4 | 指针 | 150 |
| 7.4.1 | 使用指针有哪些好处 | 150 |
| 7.4.2 | 引用还是指针 | 150 |
| 7.4.3 | 指针和数组是否表示同一概念 | 152 |
| 7.4.4 | 指针是否可进行>、<、>=、<=、==运算 | 152 |
| 7.4.5 | 指针与数字相加的结果是什么 | 152 |
| 7.4.6 | 野指针? 空指针 | 153 |
| 7.5 | 预处理 | 154 |
| 7.5.1 | C/C++头文件中的 ifndef/define/endif 的作用有哪些 | 154 |
| 7.5.2 | #include <filename.h>和#include "filename.h" 有什么区别 | 155 |
| 7.5.3 | #define 有哪些缺陷 | 155 |
| 7.5.4 | 如何使用 define 声明一个常数,用以表明 1 年中有多少秒(忽略闰年问题) | 155 |
| 7.5.5 | 含参数的宏与函数有什么区别 | 156 |
| 7.5.6 | 宏定义平方运算#define SQR(X) X*X 是否正确 | 156 |
| 7.5.7 | 不能使用大于、小于、if 语句,如何定义一个宏来比较两个数 a、b 的大小 | 157 |
| 7.5.8 | 如何判断一个变量是有符号数还是无符号数 | 158 |
| 7.5.9 | #define TRACE(S) (printf("%s\n", #S), S)是什么意思 | 159 |
| 7.5.10 | 不使用 sizeof,如何求 int 占用的字节数 | 160 |
| 7.5.11 | 如何使用宏求结构体的内存偏移地址 | 161 |
| 7.5.12 | 如何用 sizeof 判断数组中有多少个元素 | 162 |
| 7.5.13 | 枚举和 define 有什么不同 | 162 |
| 7.5.14 | typedef 和 define 有什么区别 | 162 |
| 7.5.15 | C++中宏定义与内联函数有什么区别 | 164 |
| 7.5.16 | 定义常量谁更好? #define 还是 const | 164 |
| 7.6 | 结构体与类 | 165 |
| 7.6.1 | C 语言中 struct 与 union 的区别是什么 | 165 |
| 7.6.2 | C 和 C++中 struct 的区别是什么 | 165 |
| 7.6.3 | C++中 struct 与 class 的区别是什么 | 166 |
| 7.7 | 位操作 | 166 |
| 7.7.1 | 一些结构声明中的冒号和数字是什么意思 | 166 |
| 7.7.2 | 最有效的计算 2 乘以 8 的方法是什么 | 167 |
| 7.7.3 | 如何实现位操作求两个数的平均值 | 167 |
| 7.7.4 | unsigned int i=3; printf("%u\n",i*-1)输出为多少 | 168 |
| 7.7.5 | 如何求解整型数的二进制表示中 1 的个数 | 169 |
| 7.7.6 | 不能用 sizeof()函数,如何判断操作系统是 16 位还是 32 位的 | 170 |

| | | |
|--------|---|-----|
| 7.7.7 | 嵌入式编程中, 什么是大端? 什么是小端 | 171 |
| 7.7.8 | 考虑 n 位二进制数, 有多少个数中不存在两个相邻的 1 | 174 |
| 7.7.9 | 不用除法操作符如何实现两个正整数的除法 | 175 |
| 7.8 | 函数 | 179 |
| 7.8.1 | 怎么样写一个接受可变参数的函数 | 179 |
| 7.8.2 | 函数指针与指针函数有什么区别 | 179 |
| 7.8.3 | C++函数传递参数的方式有哪些 | 183 |
| 7.8.4 | 重载与覆盖有什么区别 | 185 |
| 7.8.5 | 是否可以通过绝对内存地址进行参数赋值与函数调用 | 188 |
| 7.8.6 | 默认构造函数是否可以调用单参数构造函数 | 190 |
| 7.8.7 | C++中函数调用有哪几种方式 | 191 |
| 7.8.8 | 什么是可重入函数? C 语言中如何写可重入函数 | 192 |
| 7.9 | 数组 | 192 |
| 7.9.1 | <code>int a[2][2]={{1},{2,3}}</code> , 则 <code>a[0][1]</code> 的值是多少 | 192 |
| 7.9.2 | 如何合法表示二维数组 | 193 |
| 7.9.3 | <code>a</code> 是数组, <code>(int*)(&a+1)</code> 表示什么意思 | 193 |
| 7.9.4 | 不使用流程控制语句, 如何打印出 1~1000 的整数 | 194 |
| 7.9.5 | <code>char str[1024]; scanf("%s",str)</code> 是否安全 | 197 |
| 7.9.6 | 行存储与列存储中哪种存储效率高 | 197 |
| 7.10 | 变量 | 197 |
| 7.10.1 | 全局变量和静态变量有什么异同 | 197 |
| 7.10.2 | 局部变量需要“避讳”全局变量吗 | 199 |
| 7.10.3 | 如何建立和理解非常复杂的声明 | 199 |
| 7.10.4 | 变量定义与变量声明有什么区别 | 200 |
| 7.10.5 | 不使用第三方变量, 如何交换两个变量的值 | 201 |
| 7.10.6 | C 与 C++变量初始化有什么不同 | 202 |
| 7.11 | 字符串 | 202 |
| 7.11.1 | 不使用 C/C++字符串库函数, 如何自行编写 <code>strcpy()</code> 函数 | 203 |
| 7.11.2 | 如何把数字转换成字符串 | 205 |
| 7.11.3 | 如何自定义内存复制函数 <code>memcpy()</code> | 206 |
| 7.12 | 编译 | 207 |
| 7.12.1 | 编译和链接的区别是什么 | 207 |
| 7.12.2 | 编译型语言与解释型语言的区别是什么 | 208 |
| 7.12.3 | 如何判断一段程序是由 C 编译程序还是由 C++编译程序编译的 | 208 |
| 7.12.4 | 在 C++程序中调用被 C 编译器编译后的函数, 为什么要加 <code>extern "C"</code> | 209 |
| 7.12.5 | 两段代码共存于一个文件, 编译时有选择地编译其中的一部分, 如何实现 | 210 |
| 7.13 | 面向对象相关 | 210 |
| 7.13.1 | 面向对象与面向过程有什么区别 | 210 |
| 7.13.2 | 面向对象的基本特征有哪些 | 211 |
| 7.13.3 | 什么是深复制? 什么是浅复制 | 212 |
| 7.13.4 | 什么是友元 | 213 |
| 7.13.5 | 复制构造函数与赋值运算符的区别是什么 | 214 |
| 7.13.6 | 基类的构造函数/析构函数是否能被派生类继承 | 216 |

| | | |
|---------|--|-----|
| 7.13.7 | 初始化列表和构造函数初始化的区别是什么 | 216 |
| 7.13.8 | 类的成员变量的初始化顺序是按照声明顺序吗 | 217 |
| 7.13.9 | 当一个类为另一个类的成员变量时, 如何对其进行初始化 | 217 |
| 7.13.10 | C++能设计实现一个不能被继承的类吗 | 218 |
| 7.13.11 | 构造函数没有返回值, 那么如何得知对象是否构造成功 | 219 |
| 7.13.12 | C++中的空类默认产生哪些成员函数 | 219 |
| 7.13.13 | 如何设置类的构造函数的可见性 | 219 |
| 7.13.14 | public 继承、protected 继承、private 继承的区别是什么 | 220 |
| 7.13.15 | C++提供默认参数的函数吗 | 221 |
| 7.13.16 | C++中有哪些情况只能用初始化列表而不能用赋值 | 222 |
| 7.14 | 虚函数 | 223 |
| 7.14.1 | 什么是虚函数 | 223 |
| 7.14.2 | C++如何实现多态 | 225 |
| 7.14.3 | C++中继承、虚函数、纯虚函数分别指的是什么 | 226 |
| 7.14.4 | C++中的多态种类有哪几种 | 226 |
| 7.14.5 | 什么函数不能声明为虚函数 | 227 |
| 7.14.6 | 是否可以把每个函数都声明为虚函数 | 229 |
| 7.14.7 | C++中如何阻止一个类被实例化 | 229 |
| 7.15 | 编程技巧 | 229 |
| 7.15.1 | 当 while() 的循环条件是赋值语句时会出现什么情况 | 229 |
| 7.15.2 | 不使用 if/?:/switch 及其他判断语句如何找出两个 int 型变量中的最大值和最小值 | 230 |
| 7.15.3 | C 语言获取文件大小的函数是什么 | 231 |
| 7.15.4 | 表达式 a>b>c 是什么意思 | 231 |
| 7.15.5 | 如何打印自身代码 | 232 |
| 7.15.6 | 如何实现一个最简单病毒 | 232 |
| 7.15.7 | 如何只使用一条语句实现 x 是否为 2 的若干次幂的判断 | 233 |
| 7.15.8 | 如何定义一对相互引用的结构 | 233 |
| 7.15.9 | 什么是逗号表达式 | 234 |
| 7.15.10 | \n 是否与 \n\r 等价 | 235 |
| 7.15.11 | 什么是短路求值 | 235 |
| 7.15.12 | 已知随机数函数 rand7(), 如何构造 rand10() 函数 | 236 |
| 7.15.13 | printf("%p\n", (void *)x) 与 printf ("%p\n", &x) 有何区别 | 237 |
| 7.15.14 | printf() 函数是否有返回值 | 237 |
| 7.15.15 | 不能使用任何变量, 如何实现计算字符串长度函数 Strlen() | 237 |
| 7.15.16 | 负数除法与正数除法的运算原理是否一样 | 238 |
| 7.15.17 | main() 主函数执行完毕后, 是否可能会再执行一段代码 | 238 |
| 第 8 章 | 数据库 | 240 |
| 8.1 | 数据库概念 | 240 |
| 8.1.1 | 关系数据库系统与文件数据库系统有什么区别 | 240 |
| 8.1.2 | SQL 语言的功能有哪些 | 240 |
| 8.1.3 | 内连接与外连接有什么区别 | 242 |
| 8.1.4 | 什么是事务 | 243 |
| 8.1.5 | 什么是存储过程? 它与函数有什么区别与联系 | 244 |

| | | |
|--------|---------------------------------|-----|
| 8.1.6 | 什么是主键? 什么是外键 | 244 |
| 8.1.7 | 什么是死锁 | 245 |
| 8.1.8 | 什么是共享锁? 什么是互斥锁 | 245 |
| 8.1.9 | 一二三四范式有何区别 | 246 |
| 8.1.10 | 如何取出表中指定区间的记录 | 247 |
| 8.1.11 | 什么是 CHECK 约束 | 247 |
| 8.1.12 | 什么是视图 | 247 |
| 8.2 | SQL 高级应用 | 248 |
| 8.2.1 | 什么是触发器 | 248 |
| 8.2.2 | 什么是索引 | 249 |
| 8.2.3 | 什么是回滚 | 250 |
| 8.2.4 | 数据备份有哪些种类 | 251 |
| 8.2.5 | 什么是游标 | 251 |
| 8.2.6 | 并发环境下如何保证数据的一致性 | 252 |
| 8.2.7 | 如果数据库日志满了, 会出现什么情况 | 252 |
| 8.2.8 | 如何判断谁往数据库中插入了一行数据 | 252 |
| 第 9 章 | 网络与通信 | 254 |
| 9.1 | 网络模型 | 254 |
| 9.1.1 | OSI 七层模型是什么 | 254 |
| 9.1.2 | TCP/IP 模型是什么 | 255 |
| 9.1.3 | B/S 与 C/S 有什么区别 | 255 |
| 9.1.4 | MVC 模型结构是什么 | 256 |
| 9.2 | 网络设备 | 258 |
| 9.2.1 | 交换机与路由器有什么区别 | 258 |
| 9.2.2 | 路由表的功能有哪些 | 259 |
| 9.3 | 网络协议 | 260 |
| 9.3.1 | TCP 和 UDP 的区别有哪些 | 260 |
| 9.3.2 | 什么叫三次握手? 什么叫四次断开 | 260 |
| 9.3.3 | 什么是 ARP/RARP | 262 |
| 9.3.4 | IP Phone 的原理是什么? 都用了哪些协议 | 263 |
| 9.3.5 | Ping 命令是什么 | 263 |
| 9.3.6 | 基本的 HTTP 流程有哪些 | 264 |
| 9.4 | 网络编程 | 264 |
| 9.4.1 | 如何使用 Socket 编程 | 264 |
| 9.4.2 | 阻塞模式和非阻塞模式有什么区别 | 265 |
| 9.5 | 网络其他问题 | 266 |
| 9.5.1 | 常用的网络安全防护措施有哪些 | 266 |
| 9.5.2 | 什么是 SQL 注入式攻击 | 267 |
| 9.5.3 | 电路交换技术、报文交换技术和分组交换技术有什么区别 | 268 |
| 9.5.4 | 相比 IPv4, IPv6 有什么优点 | 269 |
| 第 10 章 | 操作系统 | 270 |
| 10.1 | 进程管理 | 270 |
| 10.1.1 | 进程与线程有什么区别 | 270 |

| | | |
|--------|--------------------------|-----|
| 10.1.2 | 线程同步有哪些机制 | 271 |
| 10.1.3 | 内核线程和用户线程的区别 | 271 |
| 10.2 | 内存管理 | 272 |
| 10.2.1 | 内存管理有哪几种方式 | 272 |
| 10.2.2 | 分段和分页的区别是什么 | 272 |
| 10.2.3 | 什么是虚拟内存 | 272 |
| 10.2.4 | 什么是内存碎片? 什么是内碎片? 什么是外碎片 | 273 |
| 10.2.5 | 虚拟地址、逻辑地址、线性地址、物理地址有什么区别 | 273 |
| 10.2.6 | Cache 替换算法有哪些 | 274 |
| 10.3 | 用户编程接口 | 275 |
| 10.3.1 | 库函数与系统调用有什么不同 | 275 |
| 10.3.2 | 静态链接与动态链接有什么区别 | 276 |
| 10.3.3 | 静态链接库与动态链接库有什么区别 | 276 |
| 10.3.4 | 用户态和核心态有什么区别 | 276 |
| 10.3.5 | 用户栈与内核栈有什么区别 | 277 |
| 第 11 章 | 软件工程 | 278 |
| 11.1 | 软件工程过程与方法 | 278 |
| 11.1.1 | 软件工程过程有哪些 | 278 |
| 11.1.2 | 常见的软件开发过程模型有哪些 | 279 |
| 11.1.3 | 什么是敏捷开发 | 283 |
| 11.1.4 | UML 中一般有哪些图 | 285 |
| 11.2 | 软件工程思想 | 285 |
| 11.2.1 | 什么是软件配置管理 | 285 |
| 11.2.2 | 什么是 CMMI | 286 |
| 11.2.3 | 如何提高软件质量 | 287 |
| 第 12 章 | 发散思维 | 289 |
| 12.1 | 设计模式 | 289 |
| 12.1.1 | 什么是单例模式 | 289 |
| 12.1.2 | 什么是工厂模式 | 290 |
| 12.1.3 | 什么是适配器模式 | 290 |
| 12.1.4 | 什么是享元模式 | 291 |
| 12.1.5 | 什么是观察者模式 | 291 |
| 12.2 | 新技术 | 291 |
| 12.2.1 | 什么是云计算 | 291 |
| 12.2.2 | 什么是物联网 | 292 |
| 12.2.3 | 你平时读的专业书籍有哪些 | 293 |
| 第 13 章 | 数据结构与算法 | 295 |
| 13.1 | 数组 | 295 |
| 13.1.1 | 如何用递归实现数组求和 | 295 |
| 13.1.2 | 如何用一个 for 循环打印出一个二维数组 | 296 |
| 13.1.3 | 在顺序表中插入和删除一个结点平均移动多少个结点 | 297 |
| 13.1.4 | 如何用递归算法判断一个数组是否是递增 | 297 |
| 13.1.5 | 如何分别使用递归与非递归实现二分查找算法 | 298 |

| | | |
|---------|--|-----|
| 13.1.6 | 如何在排序数组中, 找出给定数字出现的次数 | 299 |
| 13.1.7 | 如何计算两个有序整型数组的交集 | 300 |
| 13.1.8 | 如何找出数组中重复次数最多的数 | 301 |
| 13.1.9 | 如何在 $O(n)$ 的时间复杂度内找出数组中出现次数超过了一半的数 | 303 |
| 13.1.10 | 如何找出数组中唯一的重复元素 | 305 |
| 13.1.11 | 如何判断一个数组中的数值是否连续相邻 | 308 |
| 13.1.12 | 如何找出数组中出现奇数次的元素 | 309 |
| 13.1.13 | 如何找出数列中符合条件的数对的个数 | 311 |
| 13.1.14 | 如何寻找出数列中缺失的数 | 313 |
| 13.1.15 | 如何判定数组是否存在重复元素 | 314 |
| 13.1.16 | 如何重新排列数组使得数组左边为奇数, 右边为偶数 | 315 |
| 13.1.17 | 如何把一个整型数组中重复的数字去掉 | 316 |
| 13.1.18 | 如何找出一个数组中第二大的数 | 318 |
| 13.1.19 | 如何寻找数组中的最小值和最大值 | 319 |
| 13.1.20 | 如何将数组的后面 m 个数移动为前面 m 个数 | 320 |
| 13.1.21 | 如何计算出序列的前 n 项数据 | 321 |
| 13.1.22 | 如何找出数组中只出现一次的数字 | 322 |
| 13.1.23 | 如何判断一个整数 x 是否可以表示成 n ($n \geq 2$) 个连续正整数的和 | 324 |
| 13.2 | 链表 | 325 |
| 13.2.1 | 数组和链表的区别是什么 | 325 |
| 13.2.2 | 何时选择顺序表、何时选择链表作为线性表的存储结构为宜 | 325 |
| 13.2.3 | 如何使用链表头 | 326 |
| 13.2.4 | 如何实现单链表的插入、删除操作 | 327 |
| 13.2.5 | 如何找出单链表中的倒数第 k 个元素 | 328 |
| 13.2.6 | 如何实现单链表反转 | 329 |
| 13.2.7 | 如何从尾到头输出单链表 | 331 |
| 13.2.8 | 如何寻找单链表的中间结点 | 331 |
| 13.2.9 | 如何进行单链表排序 | 332 |
| 13.2.10 | 如何实现单链表交换任意两个元素 (不包括表头) | 334 |
| 13.2.11 | 如何检测一个较大的单链表是否有环 | 335 |
| 13.2.12 | 如何判断两个单链表 (无环) 是否交叉 | 337 |
| 13.2.13 | 如何删除单链表中的重复结点 | 338 |
| 13.2.14 | 如何合并两个有序链表 (非交叉) | 339 |
| 13.2.15 | 什么是循环链表 | 340 |
| 13.2.16 | 如何实现双向链表的插入、删除操作 | 342 |
| 13.2.17 | 为什么在单循环链表中设置尾指针比设置头指针更好 | 343 |
| 13.2.18 | 如何删除结点的前驱结点 | 343 |
| 13.2.19 | 如何实现双向循环链表的删除与插入操作 | 343 |
| 13.2.20 | 如何在不知道头指针的情况下将结点删除 | 344 |
| 13.3 | 字符串 | 345 |
| 13.3.1 | 如何统计一行字符中有多少个单词 | 345 |
| 13.3.2 | 如何将字符串逆序 | 346 |

| | | |
|--------|------------------------------|-----|
| 13.3.3 | 如何找出一个字符串中第一个只出现一次的字符 | 350 |
| 13.3.4 | 如何输出字符串的所有组合 | 351 |
| 13.3.5 | 如何检查字符是否是整数？如果是，返回其整数值 | 353 |
| 13.3.6 | 如何查找字符串中每个字符出现的个数 | 353 |
| 13.4 | STL 容器 | 354 |
| 13.4.1 | 什么是泛型编程 | 354 |
| 13.4.2 | 栈与队列的区别有哪些 | 354 |
| 13.4.3 | vector 与 list 的区别有哪些 | 355 |
| 13.4.4 | 如何实现循环队列 | 355 |
| 13.4.5 | 如何使用两个栈模拟队列操作 | 357 |
| 13.5 | 排序 | 359 |
| 13.5.1 | 如何进行选择排序 | 359 |
| 13.5.2 | 如何进行插入排序 | 360 |
| 13.5.3 | 如何进行冒泡排序 | 361 |
| 13.5.4 | 如何进行归并排序 | 364 |
| 13.5.5 | 如何进行快速排序 | 366 |
| 13.5.6 | 如何进行希尔排序 | 368 |
| 13.5.7 | 如何进行堆排序 | 369 |
| 13.5.8 | 各种排序算法有什么优劣 | 371 |
| 13.6 | 二叉树 | 372 |
| 13.6.1 | 基础知识 | 372 |
| 13.6.2 | 如何递归实现二叉树的遍历 | 373 |
| 13.6.3 | 已知先序遍历和中序遍历，如何求后序遍历 | 374 |
| 13.6.4 | 如何非递归实现二叉树的后序遍历 | 376 |
| 13.6.5 | 如何使用非递归算法求二叉树的深度 | 378 |
| 13.6.6 | 如何判断两棵二叉树是否相等 | 381 |
| 13.6.7 | 如何判断二叉树是否是平衡二叉树 | 381 |
| 13.6.8 | 什么是霍夫曼编解码 | 382 |
| 13.7 | 图 | 383 |
| 13.7.1 | 什么是拓扑排序 | 384 |
| 13.7.2 | 什么是 DFS？什么是 BFS | 385 |
| 13.7.3 | 如何求关键路径 | 386 |
| 13.7.4 | 如何求最短路径 | 388 |
| 第 14 章 | 海量数据处理 | 390 |
| 14.1 | 问题分析 | 390 |
| 14.2 | 基本方法 | 390 |
| 14.3 | 经典实例分析 | 403 |
| 14.3.1 | top K 问题 | 403 |
| 14.3.2 | 重复问题 | 405 |
| 14.3.3 | 排序问题 | 407 |
| 致谢 | | 409 |



上篇

面试笔试经验技巧篇

第 1 章 面试官箴言

第 2 章 面试心得交流

第 3 章 企业面试笔试攻略

第 4 章 面试笔试技巧

第 5 章 英文面试攻略

第 6 章 智力题攻略

面试官箴言

第 1 章

什么样的求职者能够获得面试官的青睐？求职者需要准备哪些内容来面对形形色色的面试官？什么样的企业适合自己发展？在新的工作岗位上，如何努力才能在人才济济的企业里面脱颖而出？在本章，几位资深软件工程师将现身说法，为您一一解答上述问题。

1.1 有道无术，术可求；有术无道，止于术

丁志浩，男，硕士，某知名芯片公司软件工程师。

以下这些内容是写给即将成为职业人的在校学生的，希望能够对他们的求职与以后的工作有一定的参考作用。

在介绍求职之前，我想先说一些与具体技术无关但却比技术更加重要的东西，主要有以下两个方面的内容：第一点，认清自我；第二点，保持强烈的求知欲。之所以提及这两点，并且认为它们是最重要的东西，是因为结合我的亲身经历，我认为一个人最重要的是认清自我，只有认清了自我，你才会知道自己想要做什么、适合做什么、能做什么。在某种程度上来说，这比所学的知识、技术更加重要。只有方向正确了，才会有前进的动力；有了前进的动力，才会为目标不断努力；只有朝着正确的方向不断努力了才可能会有收获。其次，要有强烈的求知欲，随着年龄的增大、个人阅历的增长，生活、家庭、工作会慢慢消磨掉你的雄心壮志，而能保持强烈的求知欲实在是难能可贵，世界上很少有学不会的东西，就看你是否用心去做了，是否愿意花时间、动脑筋、投入精力去做，万事就怕认真，只要你认真做了，通常是可以学会的。

切入正题，作为一名以程序员为职业目标的求职者，关注的领域主要还是以技术为主，IT 企业在面试的时候主要关注求职者什么方面的内容呢？以我这些年的工作经历来看，大企业看道，小企业看术。有道无术，术可求；有术无道，止于术。具体来说，大企业更加看重的是你的基础知识以及你解决问题的能力。一般而言，大企业都会有比较完备的培训机制，它可以在较短的时间内把一个什么都不会的员工塑造成一个它想要的人；而小企业则不然，他们更加注重求职者的实用性，求职者当前会什么，能给企业带来什么。这种思维方式的不同其实也是由企业的性质决定的，没有对错之分。当然这也可无厚非，所以个人建议求职者最好夯实计算机基础知识，操作系统、编译原理、算法等这些基础知识就是重中之重了，需要重点掌握。万变不离其宗，当你达到了一定程度，对你而言只是形式上的差异而已。

求职者需要如何准备才能更好地获得面试官的青睐，我觉得 IT 企业一般需要的大多数都是技术性人才，所以具有以下 3 个优点的人，一般更能受到面试官的青睐：① 基本功扎实的人，基础扎实了，以后后劲就足，发展前景就更好；② 具有强烈的求知欲、对未知领域比较感兴趣、能够接受新事物的人；③ 在某个领域有比较深入的研究的人。例如，当前好多企业都在搞云计算，如果求职者对 Hadoop 这种架构有比较深入的理解，当然就比不懂 Hadoop 的求职者成功率更高。

有了录用通知书（offer）以后，在挑选 offer 的时候，求职者往往也很纠结，其实我在这

里也不是告诉你是该选择互联网还是芯片公司，或者是其他类型企业，因为对这个问题，仁者见仁智者见智，每个人考虑的侧重点都不一样，所以在此我不给求职者说到底该选什么企业，以免误导大家，但我可以给求职者一个建议：往大的方面讲，首先是选择行业，然后选择企业，最后是选择职业。最好能够结合自己的兴趣爱好，因为兴趣是最好的老师。

入职之后，应届毕业生如何才能适应新的工作岗位，完成从学生到职业人的华丽转变呢？一般而言，刚毕业时，新人都是雄心壮志、意气风发，想在新的工作岗位上大展拳脚、有所作为，虽然这是一件非常好的事情，但是由于现代社会企业分工很明确，尤其是对于企业的新员工，刚工作时，很有可能接触的东西都是些没有技术含量或是相对边缘的东西，只是充当企业的一颗小螺丝钉而已。所以在此，我建议求职者在刚入职时，最好能够放低姿态，当将军的人，都是从小兵一步步做起的。刚毕业时的态度最重要，切记不要整天怨天尤人，否则会给人一种浮躁的感觉，对你将来的发展肯定不利。

1.2 求精不求全

褚艳利，女，硕士，某知名电子商务公司软件工程师。

时光荏苒，我已经成为 IT 业一名所谓的“老鸟”了，但我曾经也只是一名普通的求职者，在求职的路上历经风雨，但我希望我的一些经历和感悟，能为朋友们提供些许帮助。

对于应届生求职，我觉得每一场面试都是从“闻味儿”开始的。看似是一场简单的聊天，但其实求职者的各方面已经在被面试官考查了。例如，在沟通过程中，从求职者的谈吐、穿着、眼神，或多或少就能闻出很多层味道了（求职者的性格、处事态度、表达能力、沟通能力、团队合作能力）。经常会听到求职者说：“面试官今天一道技术题都没问我。”这多是面试官对求职者综合素质的一种肯定（前提是成绩单上的成绩不能太差）。如果是求取技术类职位，那么求职者的技术水平还是要积累的。

对于技术的积累，我觉得是“求精不求全”，现在的大学通常都会开设“C 语言”、“C++”、“Java”、“网络”、“数据库”、“编译原理”、“软件工程”等课程，但由于精力有限，毕竟不是每个人都可以做到门门精、样样通，所以我建议从兴趣出发，深入学习几门课程（当然，其他的课程也要学，毕竟是在技术领域，一些概念和基本原理不知晓是不行的）。例如，我个人比较钟爱数据结构、算法、C 语言、操作系统等专业知识，对这些下足工夫做足功课，曾经它们也陪着我打赢了很多场艰难战役。当然，在面试别人的过程中，我也会问到一些可能他们不太擅长的知识，如设计模式，其实我并不是希望为难他，挑他的刺，只要他能讲出自己的理解，并坦白自己这方面知识的欠缺，我也不觉得丢人，这种坦白比不懂装懂来的更真实、更有力量。所以，作为一名过来人，我觉得大部分面试官在面试时，会更加侧重于考查求职者擅长的方面，试试水究竟有多深，从这点能看到求职者未来的发展和潜力。

作为一名职场新手，在求职的准备过程中，应该根据职位要求，略作筹备。虽然说万变不离其宗，但根据职位要求，有针对性地准备一下，效果会更好。例如，面试数据库开发的求职者，DB（数据库）知识就需要好好补一下，这样不至于气氛太尴尬，也可以获得后续面试机会。对于普通的软件开发类职位，我认为求职者必备以下知识：数据结构、某类编程语言、操作系统、基本 DB 知识。

曾经我也对新人进行过面试，我认为要想获得面试官们的青睐，求职者需要注意以下几个方面的内容：

(1) 衣着装扮。对于技术类职位，衣着装扮虽然不做要求，但毕竟不能过于邋遢。女孩子画一点淡妆更好，清新怡人。

(2) 眼神交流。记着，你对面坐着的是面试官，不是墙壁，你需要跟他有眼神交流。不要怕，试着抬起头来，面试官的笑容可以缓解我们的紧张情绪，以及答不上题的尴尬气氛。害怕，其实是自己吓倒了自己。

(3) 气氛把握。语速不要太快，太快容易将自己置于紧张的状态之中。回答问题无论会不会，都要放慢节奏，你的状态直接影响面试官的身心感受以及判断。

(4) 背景了解。如果你参加一家公司的面试，最好是你真心喜欢的，并且对公司多少应该有所了解。例如，公司理念、制度、规划，谈谈你喜欢的、你认为可以改善的（这一点上要注意“度”），如果你是真的用了心，面试官往往会给予更多机会的。

(5) 轻松话题。如果谈得比较愉快，可以自己制造些轻松话题，如小吃、旅游、业界话题等。

很多时候，都有师弟、师妹们问我，挑选 offer 的时候该怎么办，需要权衡哪些内容。我不是一名职业规划师，所以不能告诉他们如何做选择，我只能告诉他们，当初我在进行选择的时候，考虑了哪些内容，以供他们参考。但总的来说，我觉得应该参考以下 5 点内容：

(1) 兴趣点。兴趣是最好的老师，如果没有兴趣，很难在工作岗位上有所作为。

(2) 公司未来的发展空间和路线。很多时候不能只盯住眼前的利益，要从长远看，一个企业的发展空间和路线、对未来市场的认知与把握都会决定你未来的发展方向，所以最好能够对企业的未来发展空间与路线有一个较清醒的认识。

(3) 薪酬福利。“钱不是万能的，没有钱是万万不能的”。一个企业再好，如果不给工资，同样没人会去，因为人要吃饭、要穿衣，所以必须仔细考虑薪酬福利。

(4) 个人成长点。每个企业对人才的定位都不一样，所以在选择职位的时候，尽量选择一些企业的核心研发部门，在这样的部门里面个人成长、个人机会都会非常好。

(5) 城市。什么样的城市是自己希望去的，是政治中心北京，还是东方明珠上海；是人间天堂杭州，还是千年古都西安；是天府之国成都，还是千湖之城武汉。各个城市有各个城市的优劣，所以没有人能够告诉你哪个城市好哪个城市不好，关键需要你自己拿主意。

其实，选完了 offer 之后，就面临着一个从学生到职业人身份的转换了。如何转换角色，我个人觉得新人初入职，最重要的就是练就基本功，这个阶段犹如蚕蜕，痛苦但却是美丽的变身。例如，我们做的是线上一级系统，承载着每秒数万笔交易的创建及支付，那么系统的架构、稳定性、容量、可扩展性、各种底层技术实现，方方面面要学的有好多，任务紧、压力大、面对着无数个不可能，这个过程看似痛苦但却让我们成长得非常之快。尤其是在项目真正上线运转起来的时候，那些你原先认为不可能做的事情现在都做到了，还做得非常漂亮，那种成就感真的是无以言表。而且做每件事情的时候，一定要把姿态降下来、心态静下来、自信提上去，与你的团队一起合作，把不可能当做为历史，把可能写在今天。经历一段时间的洗礼之后，仔细思考一下，问一问自己是否可以独当一面，是否在业界，至少在公司部门内，可以听到你的声音，可以看到你的建议。如果可以，那么恭喜你，你应该可以升职了。

1.3 脚踏实地，培养多种技能

廖兰新，男，硕士，某创新型企业高级研发工程师、开发经理。

作为一名一线的技术研发人员，结合自己多年在技术上的经历，在此分享一些经验给即将走入职场的应届毕业生，帮助他们在人生的路上少走一些弯路。

(1) 行业选择。在应届毕业生进行择业的时候，我个人觉得选择适合自己的行业是非常重要的。对于计算机类专业的毕业生，可供选择的行业很多，如商业银行类、国企、央企、传统的软件公司、新兴的互联网公司。而这些行业又各有各的特点，对能力的要求迥异。例如，国企普遍工作轻松、薪资一般（体制内）、福利很好，对技术要求不是太高，对项目进度的要求一般不紧迫；互联网公司工作一般比较辛苦，对项目进度要求非常紧，技术研发能力也要求高，而企业文化一般较为自由，其薪资待遇一般比较高。所以，求职者应该根据自己的兴趣爱好以及能力特点选择合适的行业。

(2) 技术领域选择。随着现代化管理技术的不断发展，IT 企业中的技术分工也越来越明显。俗话说：“隔行如隔山”。同样是计算机科学技术，不同技术领域的人在技术上也是非常迥异的，如互联网企业与芯片企业关注的重点就不一样。对于应届毕业生，一般也很难做到“通才”。所以，在求职的时候，尽量选择自己喜欢的专业领域或者自己擅长的专业领域，这些会决定你后面的职业生涯的主要工作内容，而且一般也不会轻易更换。

(3) 雇主选择。不同的雇主对求职者的要求也不一样，以科技巨头公司与创业型科技公司为例加以比较。创业公司一般研发人员相对较少，每个研发人员都需要能够独当一面，对整个产品的核心代码都了如指掌，上至前端开发、Web 界面，下至后台底层实现、操作系统，所以这对于个人成长是非常好的锻炼机会，但同样，创业公司也有其自身的局限性，由于工作的需要，员工一般身兼数职，经常加班，而且在专业技能上都不够规范，相比大型科技公司完善的团队、严格的规章制度等，相对欠缺。

但总的来说，在创业公司，更能够全方位地激发个人潜能，多角度地发展个人能力，大公司可以小而精地锻炼某项专业技能。当然以上的说法也不是绝对的，比如某些小型高科技公司也聚集了业内的人才，完全具备大企业的“高精尖”特点，而一些大公司的某些部门在初创阶段可能也会像创业公司一样艰苦。如果你决定不了，那你就尽量去一家步入正轨的大公司。

(4) 求职建议。因为企业需要，我曾经担任过一段时间的面试官，帮助招聘企业新人。我们确实非常希望招到优秀的人才，但在招聘的过程中也遇到了很多很遗憾的事情。例如，有的人在面试的时候因为紧张或是其他原因，真实才能发挥不出来；有的人水平一般，却夸夸其谈，不脚踏实地，真的让设计算法时，一头雾水。在此，我想说明一点，企业在招聘的时候，需要这样的人才：对人对事有信心、掌握多项技能、基础扎实、有冲劲、愿拼搏。所以，我建议应届毕业生在平时的学习中，一定要脚踏实地地学好专业知识，适当地扩展专业技能。

(5) 能力培养。进入工作岗位之后，很多应届毕业生迷茫了，很难从学生的角色向职业人的角色转变，我觉得计算机职业人应该注重培养自身的3种能力：技术能力、管理能力、领导力。职业新人往往依赖技术能力进入职场，最初的晋升也主要来自技术能力，它可以让你成为一个优秀的单兵和一个称职的经理，但很难成为优秀的经理人，因为它的杠杆效应非常有限，这就需要第二种能力：管理能力。管理其实是对资源的管理和利用，以有效、可靠地生产产品或提供服务。人的大多管理能力都可以学习到，教育、经验、培训都可以提高管理能力。当然人的悟性也很重要，能够从表面现象中分析出规律，对管理能力来说很重要。管理能力主要是释放物的能力。它可以给你一定的杠杆力量，能让你在小范围内有所贡献，但不会让你走很远。这时候就需要第三种能力：领导力。领导力是释放别人的能力，再通过别人来释放个人或

物的能力。领导力巨大，是因为它有二级杠杆的效用。对于领导，技术能力的重要性非常有限，管理能力次之，领导能力最重要。不要认为职业道路是单行道，即从技术职位向管理职位过渡，再由管理职位向领导职位过渡。

1.4 保持空杯心态

王震，男，硕士，某知名互联网企业研发工程师。

好友何昊拜托我一件事情，就是给当前程序员写一些关于求职的意见与建议，这着实有些为难我，并非我不愿意去做这件事情，而是因为本人入行虽然比较早，但入职却不太久，与一些资深的 IT 们相比，也只能算是初出茅庐，所以不敢妄自尊大，不过可以分享一下本人这些年来的几点粗浅体会，以起到抛砖引玉的效果。

程序员，作为以技术主打的 IT 专业从业者，对于个人的发展，扎实的基本功将更有利于在行业里站稳脚跟，走得更远，发展前景也更加明朗。“术业有专攻”，所谓专业，在于求深而不在于求广。当然，话也不能绝对，更广的知识面可以帮助你对整个大行业背景有一个比较清晰的认识，知道自己处在产业链中一个什么样的位置，能够做出多大的成就，能够有多大的发展空间。结合我自己的经历，以软件类研发为例，具体而言，后台开发方向，系统、网络的底层比如操作系统事件机制（如 Windows 消息机制、Linux epoll 等）、TCP/IP 协议栈、C/C++ STL 等，这些是服务器开发的主战场，对这里每项技术需要了解的程度就如同战场上你对手中所握兵器需要熟悉的程度一样，也许对小规模服务器程序开发而言，谈论这些内容可能有些夸大其词、危言耸听的感觉，但确实存在很多需要如此考虑的情况。例如，当前很多网上订票系统的性能就很难满足实际应用的需要，引起用户的极大反感。而在前端方面，由于技术更迭较快，对于程序员而言，快速学习能力就显得尤为重要，紧跟时代潮流就要看准当前的形势，了解站在时代前沿的人有哪些，他们做了什么，即他们的研究成果有哪些。

至于经典的数据结构、算法，其实无论是前端研发还是后台研发都会有所涉及，不过更深入的掌握一般也只在较专业的算法密集型领域，如搜索、GIS 等。而对于你、对于面试官更注重什么，则看你们更侧重哪方面的内容了。

对于已经入行的程序员应聘新的企业，即通常所说的跳槽，经验及能力通常是面试官考查的重头戏。不像刚毕业的学生，白纸一张，面试官还会考查一下你的学习能力或个人发展潜力。说得再直白一点，作为利益链条上的一环，你具备什么资本，能为公司创造什么价值，才是面试官关注的焦点所在，这也是你需要真正搞清楚并且为之准备的内容。所以做过什么项目，取得什么样的成就，既说明了你的过往表现，也能对你的潜在价值表露一二。

进入工作岗位，我相信，不管是刚入职的毕业生还是已打拼多年的程序员，以空杯心态去融入当前企业文化，绝对不是件坏事。只有认可了你的雇主，工作之时，你才能积极主动，才能上进、才能提升。职业发展方面，一般公司都会有量化的绩效指标，在完成这个指标的同时也是对自己的一种提升，而在任务指标之外，结合自身情况制定出半年或全年个人发展规划，可以说是对自己短期能力提升的督促和目标实现的指引，有助于自己向着更明确的方向发展。

以上愚见，称不上是成功的经验，只是我这么多年对程序员这个行业一点浅薄的理解而已。

1.5 职场是能者的舞台

林方超，学士，北京某上市公司软件工程师。

关于应届毕业生如何求职这个问题，老实说，我的“经验”并不是很多，若干年以前，因为应聘前准备的比较充分，所以命中率比较高，虽然也拿到了几个不错的 offer，但最终还是选择了现在这家企业。这么多年过去了，一路走来，感悟颇深，回过头来看当初求职这件事情，也是回味无穷。我认为一个非常有针对性的准备工作，包括心理准备与知识准备，对于计算机相关专业应届毕业生求职非常有用。

首先，作为求职者，应当找准自己的位置，即通常我们所说的职位。一个对职位有着准确预期、对自己有着准确定位的人，在个人简历、面试中都能够表达出更准确、更吸引人的信息，而不至于投递完简历之后就杳无音讯。而找准一个方向，找准一个行业或是锁定一个企业，不仅可以缩小求职的范围，而且还可以让你在有限的精力、有限的时间内将准备的内容进一步深入，进一步细化。如果你做到了这一点，不管是大企业的招聘还是小企业的招聘，也不管是在笔试还是面试的时候，你很快就能发现一件事情，就是真正能够与你竞争的人、能够把你比下去的人真的是屈指可数，此时你就成为了求职大军中笑到最后的人（插入一个感悟：时下流行的技术往往被人普遍提及，反而是陈词滥调，只有真正理解其中思想的人才能脱颖而出，如果没有十成的把握，我宁可绝口不提）。

通过一些有针对性的准备工作后，笔试一般就不会存在问题了。而紧接着需要面对的就是面试这一关，每一次求职的机会都很宝贵，每一次面试的机会也很难得，而成功随时就会降临，作为求职者，不应当将机会随意浪费掉，将成功拒之门外。所以，不要总以为自己运气好，可以“裸装上阵”赌一把。因为作为求职者，在与企业的博弈中，我们是弱势的。因此，你需要对求职的企业以及岗位有一定的认识与见解，当然，你通常在此之前对其可能一无所知，很迷茫，如果此时稀里糊涂去了，也自然是稀里糊涂回来。其实，只要提前做好功课，这些问题都称不上是问题，因为稍微有点名气的企业都会有自己的宣传网站，在这里面会详细地介绍企业的发展历程和现状，此外不少网站在校园招聘的同时也会列出详细的招聘信息，这些内容都可以好好看看。至于对这些内容需要了解到什么程度，就看这家企业在你心中的地位了。想象一下，在面试的时候，当你谈到许多他们公司的一些信息的时候，面试官会想要给你介绍更多，甚至想带你去实地参观一下，那么接下来基本就可以直接谈待遇、谈工作了。

介绍了再多的理论和方法，也只能说是“纸上谈兵”，是否可行还需要行动来验证，只有行动了才能体会到其中的价值。如果成功地拿到 offer，那是最理想的；如果没成功，最好要让面试官给你些建议，遇到说不出来或闪烁其词的情况，说明面试官是凭个人喜好作出的判断，大可不必理会；而一针见血的评价以及善意的建议都会对你未来的求职、成长有很大的帮助，所以不能被一根绳子绊倒两次，无论是成功了还是失败了，都要有启发，成功可以收获经验，失败同样可以得到教训。

挑选 offer 也是一件比较艰难的事情。个人建议，最好按照自己的职业规划进行比较，但如果自己确实没有很明确的职业规划，或是从来没有想过职业规划这个问题，你可以优先挑选有发展潜力的工作，这样的工作会给你带来许多意外的收获，最终推进你形成自己的职业路线，构建你的职业规划。

最后，我想说的是职场是能者的舞台，真正比拼的是各种能力。技术是一种能力、交际是一种能力，发挥好任何一种能力都能使你的工作如鱼得水、锦上添花。因此进入工作岗位后该

如何发展，并非我一两句话能够回答的了的，关键还是看各位自己，“八仙过海各显神通”了。

1.6 学会“纸上谈兵”

卢山，硕士，某知名搜索类公司软件工程师。

2009 年硕士毕业于中国科学院计算技术研究所，到目前为止换过两次工作，最终选择了现在的这家企业。作为一个职场的过来人，经历了很多事情，有初出茅庐时的意气风发，也有历经沧桑后的冷静思索，在这里我谈谈技术类职位面试应该怎样准备的问题。其中有一些建议，也是与产品类面试相通的。

在谈论面试笔试如何准备前，首先我想说一些求职者在应聘的过程中的常见误区。一是认为 GPA（成绩）越高，则面试成绩越好；二是认为编程的技术越好面试成绩越好；三是认为在纸上写代码与在计算机上编程是一样的，不用准备或是不用特殊准备。我个人觉得，这些理解都是片面的。事实上，虽然说面试是一种主观行为，但它也是一种考试，准备的因素占了 50% 以上。但它又不同于高校中的考试，因此与 GPA 关系非常小。

既然准备如此重要，那么求职者就要做好读技术面试书这个环节的准备。此类书籍非常多，每本又很厚，要怎样在有限的时间内在众多的考点中识别出面试官常问的那些问题呢？规律是有的，因为面试官们精力有限，很少去凭空想象一些题目，很多都是套用现成的知识点，所以不论你应聘什么职位，考点总会以这样的规律出现，复习中遇到就要记住。一般情况下，需要注意以下几个方面的内容：

（1）列举处常考。在复习时看到一个知识点分成几个项目列出来的，就很可能是要考的。例如，“在网页中使用 CSS 有 3 种方式，inline、internal 和 external”。

（2）比较处常考。例如，“C 中的 auto, static, register 和 extern 的区别是什么？”、“const 与 define 有什么区别？”、“C++ 中 struct 与 class 有什么区别”等。

（3）性能优化常考。例如，“怎样提高网页加载速度”，“如何提高数据库查询效率”，内存泄漏的原因、识别及防范等。在 C 语言、Java 语言和算法方面也会经常考到类似的问题。

（4）算法设计与实现常考。经常会针对某些特定的算法对求职者进行考察，同时时间复杂度也很容易考，所以求职者要在掌握好算法原理、代码实现的同时，记住它们的复杂度。

除掌握常考的考点外，你还要练习在纸上写程序。脱离了功能强大的 IDE（Integrated Development Environment，集成开发环境），在纸上写程序就与在计算机上非常不一样了。这里没有自动提示，没有语法高亮，没有拼写纠正，没有自动编译、链接与运行，全凭你平时写代码的积累了。但是在笔试和面试中，常常要当场“纸上谈兵”，如果不熟练就要吃亏，所以这一关必须要过。

1.7 小结

尽管每一个面试官的工作背景不一样，个人能力也不一样，而且面试套路也可能独具匠心、别具一格，但是，他们的目的只有一个，发掘最适合企业的优秀人才。对于求职者而言，面试官的喜好往往决定了求职者的去留，所以面试官的意见与建议，求职者应该好好斟酌，认真体会，从而不断地提升自己，成为每一个企业争抢的“千里马”。

面试心得交流

第 2 章

“前车之鉴，后事之师”。本章以各大名牌高校、研究所的应届毕业生的亲身求职经历与体会为蓝本，对当前程序员面试笔试相关的准备工作、时间计划、书籍阅读、面试技巧、offer（中文指录取通知）选择等多个方面的内容进行了独到地分析，对于未出校门的应届毕业生有着极大的指引作用。

2.1 心态决定一切

董哥，男，中国科学院计算技术研究所 2012 届硕士研究生，现就职于北京腾讯搜搜。

1. 抛砖引玉

找工作的过程是综合实力较量的过程，一个好的 offer 背后凝聚着无数辛勤的汗水，需要勤奋、坚持、积累、付出。这里介绍一下自己找工作的经验，希望对师弟师妹们有所启发。需要注意的是，完全做到了这里提到的几点并不意味着你一定可以拿到一流的 offer，我仅是抛砖引玉而已，如果想在找工作时得心应手，需要平时不断积累和总结，领悟其中的真谛。

2. 心态决定一切

对于找工作，心态很重要。找工作之前，一定把心态端正。20 年寒窗苦读，最重要的一个目的是找一份理想的工作，从而实现自身的价值，因而我觉得，我们至少应该像准备高考那样，全身心地投入到找工作的准备中，将之前所学的知识重新温习整理，以便将所有能力能够最大限度地发挥出来，进而向面试官充分展示自己、推销自己。

3. 冰冻三尺非一日之寒

关于找工作前的准备，有两个因素直接决定着你是否能最终被录用，一个是项目，另一个是基础知识，这两者中任何一个被面试官相中，均可能拿到 offer。

对于项目，不在多而在精，一般的项目，如普通的管理系统、网站等，面试官几乎不用耗费脑力，一眼就能看到底，没有什么好讲的，最切合也最能引起面试官兴趣的项目往往是与他现在的领域相同或相近，解决的问题的确具有一定的难度且提出的解决方案具有一定的创新点。但遗憾的是，对于大部分毕业生，项目的深度往往不够，毕竟想在研究生短短的两三年时间里成为这方面的专家，还是比较有难度的，所以这个时候就全靠你的基本功了。

基本功大致可分为以下几个部分：编程语言，数据结构与算法，操作系统和其他小知识点。对于编程语言，个人认为 C 语言是必须掌握的，很多公司把 C 语言作为必考项。另外，要在 C++ 和 Java 两种面向编程语言中选一个，主要知识点是面向对象编程中的一些基本概念，如虚函数、构造函数、析构函数、拷贝构造函数等。有一些题目已经成为经典，是必须、一定要掌握的。例如，（C++ 语言）虚函数是怎么实现的？构造函数可以是虚函数吗？为什么鼓励将析构函数设计成虚函数？对于数据结构和算法，这是面试的重点，很多公司基本上只考算法与数据结构，这就需要大家平时多积累、多练习。尤其对一些基本数据结构和算法，要非

常清楚，如单链表反转、Trie 树、两个数组交并差集等。对于操作系统，主要掌握 Linux 里的一些基本概念，如线程、进程、内存管理、文件管理等，这些也会在面试中出现，一定要好好复习。最后是一些其他知识点，如设计模式（单例、工厂模式等）、编译原理（程序从编译到运行要经历的几个过程）等。

4. 修炼程序员之“葵花宝典”

找工作的过程中，一些经典的题目，一定要反复推敲，很多题目来自固定的几本参考书，大家应该好好琢磨一下这几本书中的题目。

(1)《编程之美》。这是一本实战书，任何找过工作的人都知道，很多笔试面试题直接来自该书，值得各位找工作的应届生认真地阅读和讨论。此外，该书中有些题目难度过大，从找工作的角度考虑，可暂时不看。

(2)《编程珠玑》。该书主要介绍软件设计思想，书中的例子已经成为百考不厌的经典题目，如数组循环移位、随机采样算法等。

(3)《算法导论》。该书对各种常见算法有很深入的讲解和详尽的证明，并对每个算法的起源、动机和求解过程有较多的涉及。

(4)《深入理解计算机系统》。该书从程序员的视角介绍了计算机系统。几乎囊括了计算机的各类技术，包括数据表示、C 程序的机器级表示、处理器结构、程序优化、存储器层次结构、链接、异常控制流、虚拟存储器和存储器管理、系统级 I/O、网络编程和并发编程等。该书中提到的一些知识点，常作为面试题目出现，如 Linux 信号量、虚拟内存管理等。

5. 八面玲珑

关于找工作的技巧，主要介绍两点，一是回答问题的技巧。对于项目，主要回答点应该是遇到的挑战和解决问题的思路，对于算法问题，要从复杂度高的算法逐步向复杂度低的算法过渡，第一眼见到题目，可先将自己想到的思路说出来（如 $O(n^2)$ 复杂度），然后不断优化（如 $O(n \log n)$ 复杂度），最后尽量得到一个最优的算法（比如 $O(n)$ 复杂度），这时候可能要在纸上写出来，一旦没有了思路，应该主动要求面试官加以提示。另一个是交流技巧，这里指的是面试者之间的交流，这一点非常重要，每当前一个面试者面试完后，应该主动跟他交流，主要询问一些个人收获和心得，尤其是别人的失误，应该尽量避免，因为面试官一天要面试众多的求职者，很可能会对不同的求职者提出相同的问题。

6. 多多益善

最后是 offer 的选择。offer 尽量多拿一些，以便给自己留一些选择的余地，至于怎么选择 offer，这是个人的问题，每个人的侧重点不一样，因人而异，但我觉得适合自己的就是最好的，没必要和别人进行比较。

2.2 假话全不说，真话不全说

萧叶，中山大学 2012 届硕士研究生，现就职于睿初科技（深圳）有限公司。

1. 万事趁早

我大概是研究生三年级新学期开学后开始准备找工作的，从后来的情况来看，我已经准备晚了，因为校招时间提前了半个多月。这也给了我一个教训：万事趁早，因为我们不能预知公司什么时候来招聘，只能自己提前做准备。准备太晚的结果就是 9 月中下旬的阿里巴巴、淘宝等公司的招聘全没赶上。

2. 不经一事，不长一智

虽然我找工作的两条原则很早就确定了：去外企、搞技术，但是当校招开始时，我几乎还是逢公司必投简历（当然是软件研发类的），一来是因为自己手头无 offer，总是有些没底，不知道自己是否能够找到满意的工作，特别是看到周围暑假实习回来就拿到 offer 的同学，心里不免更加担心，紧迫感更加强烈；二来是因为本科毕业就直接读研了，没有真正找过工作，对找工作还是很陌生，虽然有师兄师姐留下的一些找工作的心得和建议，但毕竟“绝知此事要躬行”。

所以我认为，“海投”也没有什么错误，虽然“海投”的这些公司并不都是自己非常想去的，但是如果不趁早积累和总结一些属于自己的找工作心得，等到心仪的公司来时胜算的把握有多大就很难说了。

3. 读书破万卷，面试如有神

因为找工作准备得比较晚（我觉得从暑假开始准备算是比较适合的），所以我基本上是一边找工作一边准备面试笔试，而准备主要就是看书。对于大多数没有项目经验或项目经验少的研究生和本科生而言，看书是投入产出比最高的找工作准备方式。因为笔试面试最常见的内容就那些：语言、数据结构与算法、操作系统、软件工程等内容。语言类靠编程指南之类的书籍即可，其他专业知识点我认为比较有帮助的书籍有《(more) Effective C++》、《(more) Exceptional C++》、《C++ Common Knowledge》、《算法导论》等。语言类书籍给出的都是语言规范等确定性的知识，告诉你是什么，非常适合应付笔试；而后一类书则好比内功心法，给出一个场景，分析各种方案的优缺点，告诉你为什么是这样，看这类书的收获，与编写的 C++ 代码量正相关，面试时有水平的面试官比较喜欢问这类问题。这系列的书，无论读者水平的高低（当然基本语言知识得懂），总能从中领悟到一些东西，而且每次再读，又有新的体会，不仅仅适合找工作时读。至于算法方面，我认为这不是看看书突击一下就可以显著提高的，就算把那些常被问到的排序算法死记下来，面试时也不太管用，这个还是要靠平时的积累和悟性。

4. 人性化的简历

简历的制作上，排版可以讲究些，目标是让筛选者快速、准确地找到他所关注的内容（如技能、项目经验、成绩等），以两页为宜（有人说最好一页，但是我感觉一页根本写不下，也容易让筛选者觉得材料有点单薄）。至于打印，我觉得最好选稍厚一些的纸，至少不能很清晰地看到背面。总之，要让简历的筛选者拿着、看着觉得舒服。彩色打印就不必了（明确要求的除外），除了相片是彩色的，和黑白打印并无大异。

5. 假话全不说，真话不全说

面试到了尾声时，面试官（通常是技术主管、人力资源或经理）有时会问有关求职者职业规划、家庭背景、已经拿到了哪些 offer 等情况。尽管在此之前，有很多师兄师姐给我传授了相关技巧，但是我还是按照自己的真实想法来回答，也许正是因为自己太“老实”的原因，最终与几个公司擦肩而过：华为、爱立信都问了我拿了哪些公司的 offer，我如实回答了，还有一家公司问我如果给我 offer 我是否签约，我说要考虑一下。我觉得实话实说并没有什么不当，人的本性都是差不多的，一个对自己负责的毕业生找工作时货比三家，最终选择自己最满意的工作是无可厚非的，自信的企业应该能够理解这一点。但实话实说也并非一定要回答面试官的所有问题，有一位面试官问我家里的情况问得过于详细，还有两个问到了其他公司给的待遇问题，我都没有正面回答。拒绝回答问题就要靠技巧了，要尽量委婉地拒绝，不要太过直接，伤害彼此的感情。

6. 豆腐白菜，各有所爱

对于 offer 的选择，这是一个见仁见智的问题，自己最满意的就是对自己来说最好的。我找工作时主要有两条原则：第一，以外企为重点，希望将来有机会到国外工作，但也并不是非外企不去。第二，非技术类的工作不做，因为我知道自己不适合也不太喜欢做售后、策划等工作。结果，拿到的几个 offer 中，爱立信和睿初都算是符合这两条要求的。在满足条件的这两家公司中，爱立信给予的是带附加条件的 offer，要求现在能够过去实习至少两个月，人力资源和项目经理先后打电话问了两次，看得出来是确实急缺人手而不是为了赚廉价劳动力，但是导师不同意实习，所以只好放手。而深圳睿初科技是我找工作以来所有公司中流程最严格（1 轮笔试，1 轮电话面试，4 轮现场面试，两轮总部的电话面试）、最人性化的一家公司，我对它的期望和好感就是在一轮又一轮的面试和沟通中不断提升的，以至于当它最终给我 offer 时，我毫不犹豫就签了。

其实我觉得首先得确定自己找工作的原则，明白什么是自己最为看重的，然后重点准备符合自己原则的那些公司的笔试面试。

2.3 走自己的路，让别人去说吧

小郭，女，西安电子科技大学 2012 届硕士研究生，现在计算机网络与信息安全教育部重点实验室攻读博士学位。

这是我第一次找工作，现在把自己找工作的一些情况以及心得整理出来，一来对自己的经历作一个总结，二来可以为师弟师妹们提供一些信息。本人本科专业计算机科学与技术，毕业后直接保送了本校的计算机软件与理论专业读研，研究生阶段从事的基本都是软件类研发工作。

1. 无悔的选择

在研二时，我就开始纠结于找工作还是继续念博士之间，但紧迫感不够。到了研三，不能再犹豫了，我做的决定就是先找工作，看看找的情况，毕竟找工作是一份很宝贵的经历。对于工作，我真正拿到了 4 个 offer：华为的软件研发、阿里云的无线平台开发、百度的客户端研发和腾讯的后台研发。但最后我还是选择了攻读博士学位。

2. 出师未捷身先死

我是从研二放暑假回学校后开始着手找工作的，应该算比较晚的，复习的内容其实就是面试指南、《编程之美》和各种专业课书（如数据结构、操作系统、计算机网络等）。现在校招的时间越来越早，当第一批公司来的时候我还有很多内容没有复习。

来的最早的公司是联发科，毕竟是第一次找工作，当时我心里还是很紧张的，笔试题不算难，我顺利过关了。接着就是一面了，一面大概半个小时的样子，主要问的就是实验室做的项目，一面结束后等待二面消息，可是当身边很多同学都收到二面通知时，我却依然没有收到二面通知，第一次找工作就碰壁，当时对我打击还是挺大的。后来我静下心来总结了一下这次面试失败的原因，其实联发科问我的问题并不是特别高深，都是一些基础知识，失败的主要原因我觉得在于两点：第一点是面试太紧张；第二点是准备不充分，尤其是项目部分，与面试官的沟通不是很好，面试官对我做的项目应该没有什么了解，而我又没有提起面试官的兴趣，因此我说的话面试官不懂，面试官问的问题我也没有清楚明了地回答。

3. 过五关斩六将

接着9月下旬华为、中兴等公司陆续开始了校招。华为面试的场面非常壮观，每天参加面试的学生数以千计，4轮面试不停，我是从下午一点开始面试的，第一天直到晚上九点才面了三轮，而第四轮面试需要等到第二天，于是我拖着疲惫的身体返回学校。华为的面试一共分为4轮，分别是：技术面试、机试（上机编程）、性格测试和HR（人力资源）面试。技术面试就问了一下实验室项目然后让写了个简单的程序就通过了，接着是上机测试与性格测试，机试并不是要求编写的程序完全正确了才让通过，而是根据写的程序进行打分，然后参照同一批人的水平来决定是否通过，其实通过率还是挺高的。而最关键的就是性格测试了，很多人都在性格测试这一关止步了，实在可惜，我一个同学就因为性格测试的时候仔细斟酌，害怕回答得不好，最后没有通过性格测试。对于性格测试，我的心得就是不要太紧张，放轻松点，做题前后要保持一致，尽量不要前后矛盾，按自己的真实想法耐心回答即可。第二天进行的第四轮面试其实也只是随便聊聊天，面试官就问了一下我的家庭背景以及一些与技术无关的问题，接着就直接发给我口头 offer 了。

之后是百度、腾讯和阿里云三家互联网公司，我感觉百度最注重算法，面试时间也最长。其实能拿到这3个 offer 我个人觉得很很重要的一点就是心态，我去面试这三家公司的时候心里很放松，没有一点紧张，我就权当是去锻炼锻炼，这样效果反而会更好。当然也不是只要不紧张就可以了，面试成功的因素是多方面的，与你碰到的面试官，当年的就业形势都有很大的关系。但是在自身方面，除了心态好，还有就是要充分的准备，尽量把自己会的面试官也感兴趣的东西告诉面试官。在项目方面因为有了之前面试的经验，我在与面试官讨论项目的时候越来越熟练，对项目的理解与总结也越来越好，因此讨论项目这一部分我的问题越来越少，而且我个人认为不仅要对自己做过的每一个项目做充分准备而且一定要实话实说，因为每家公司注重的与感兴趣的内容不同，或许他们会对你没有准备的项目很有兴趣，如果这时候你显得很生硬，那么就对你不是很有利了。在我面试期间，阿里云对于我曾经参与过的与编译器有关的项目很感兴趣，而百度则对网络安全中的身份认证感兴趣。当然，实话实说的意思是不能说假话，但是并不意味着要把所有实话都说出来。如果说假话被面试官拆穿了，那么就彻底没戏了，有时候可能有人会抱着侥幸心理，不过我碰到的这三个公司的面试官对我简历上写的项目总有一个会很熟悉，有的甚至不止熟悉一个，因此还是踏踏实实、实话实说比较保险。这三家公司的面试题与华为、中兴的区别甚大，他们更注重的是你的能力和反应，一个问题面试官可能会与你讨论很长时间，如果很顺利地回答好了，那么面试官会将这个问题延伸，如果不能回答出来，面试官会给你提示并且与你讨论。总之你和面试官交流的过程就是把自己的能力展示给面试官看的，就算回答不出来或者答得不完美其实也没有很大的关系。

4. 成绩第一

除了心态好、对项目熟悉之外，就是技术了。我在面试过程中，虽然没有把面试官问的问题全部回答出来，但是也差不多，因为有很多面试题涉及的知识都是我以前在实践中或者在技术书籍中看到过的，在研究生阶段我利用课余时间看了不少专业书籍，如《编程之美》、《编程珠玑》、《计算机程序设计艺术》、《Windows 程序设计》、《C 陷阱与缺陷》、《C 专家编程》和《深度探索 C++对象模型》等。这些书籍对我找工作的帮助非常大，不仅仅是面试题中可能会出现，考虑问题的思路或者是方法都可以从书中得到启发。

在面试过程中，笔试成绩高还是很有优势的，我在阿里云面试的时候就是得益于笔试成绩很高（后来面试的时候看到的，接近满分），面试过程中面试官貌似对我很有信心，没有特别为难我，尤其是第三轮面试的时候，部门领导直接说已经可以确定我通过了，整个过程中都说

我的笔试成绩很好，因此没有问什么技术问题，都是给我介绍他们的工作情况。

拒绝我的 Marvell（美满）上海研发中心是一家全球领先的半导体厂商，因为是外企，因此他们对英语有比较高的要求。我很早就投了这家公司的简历，过了很久之后才接到了他们给我的电话，让我去面试，一面结束后我才了解到之所以让我去面试是因为我有参加 ACM 竞赛的经历，所以在这里插一句，有机会的话一定要尽量多参加一些竞赛，一来可以锻炼自己的能力，二来可以结识一些不错的同学，而且有可能会让你拥有比别人更多的机会。Marvell 的面试一共有三轮，三个面试官全都问技术，三面下来花了四五个小时，面试官不同于上面提到的那三家互联网公司那种随和的感觉，每个人都很严肃、很犀利。第一面主要问我算法，让我设计一个两部电梯的调度算法，主要从人性化的角度去考虑，我设计了几个方案之后面试官都不太满意，算法题结束之后又用英语交谈了一下，第二面的面试官主要问的是与项目有关的内容，还有一个与专业无关的测试，问项目的时候问得非常细致，幸亏来之前有所准备，这些结束之后他让我说说如果让我测试一款手机我会怎么测试，越完整越好，由于在此之前我曾去中兴西安研究所参观过手机测试部门，所以说了一些自己见到的，面试官对我的回答结果应该还算满意。第三面的问题包罗万象，软硬件都有所涉及，软件我还能应付，硬件就有些力不从心了，因为研究生阶段我都没有接触过硬件。当天面试完毕之后我感觉应该没戏，不料过了一段时间我收到了 Marvell 美国那边的邮件，叫我把 GPA 和英文简历发给他们，当时我已经决定上博士了，还在准备英语考试，所以就没在意，随随便便发了一下，之后就没回音了，我感觉是因为英文简历不过关。那份英文简历是我在暑假的时候草草做的，没有修改，很多地方都不完善（甚至有语句不通的可能）。虽然没有收到 Marvell 的 offer，不过我的收获还是很大的，这次面试完我知道了自己的知识和水平还有很大的提升空间，只有以后再努力了。

5. 走自己的路，让别人去说吧

最后我拒绝了所有的 offer，选择了继续攻读博士学位，这里有一些主观原因，也有客观原因。总之，选择了就要走下去，其实每个人都会在生活中遇到很多选择，我觉得不管你选择了什么，只要是你自己的选择就不要后悔，踏踏实实地走下去，坚持是最重要的。

2.4 夯实基础谋出路

jololee，男，浙江大学 2012 届硕士研究生，现就职于杭州网易游戏。

1. 万事不备

我是从 7 月份开始准备找工作的，刚开始并不算太努力，断断续续，自己也比较松懈，所以只是零零散散地进行着复习，对于知识点的掌握也并不是非常精通。直到 9 月份重心才完全投入到找工作中，开始看一些专业书籍，如《算法导论》、《C 专家编程》等。

2. 夯实基础谋出路

对于面试笔试的准备，我觉得基础是根本，所以需要多学习一些基础知识，参考的图书有《算法导论》、《数据结构》、《深入 Java 虚拟机》、《Java 多线程模式》等，其他的内容由于时间关系，看得比较匆忙，如《编程之美》、《编程珠玑》。另外，编程指南类速成书籍也看过，不过感觉一般，应付小公司还可以，真正的大公司，仅凭这种书肯定是不够的。所以我的教训是如果时间允许，多读书，读好书，夯实基础是王道。

3. 字字珠玑

找工作的过程真是道路崎岖，现在分析关键原因还是准备得太晚，9 月份第一波招聘潮到

来的时候，我还没有看过《编程之美》等书籍，这也导致我错过了一些好公司。

简历制作要区分国企、私企、外企3种，国企考查求职者的综合素质，需要经历辉煌，他们一般更注重综合素养，而不仅仅是技术细节；而私企一般会深入追究，需要把简历上的每个项目弄清楚，技术要扎实深入；外企需要能够用英语讲述自己的经历、说清楚一个项目的工作以及有良好的表达能力。

不准备算法、错过一半公司；不准备项目经验和技能，错过另一半公司。如果去外企，英语是必需的。

4. 多方询问

一般可以从师兄师姐那里得到一些关于企业的详细资料，也可以从学校bbs（水木清华、饮水思源、飘渺水云间、西电好网、北邮人等）上的帖子获取相关信息，还可以广泛征求同学或朋友的意见和建议。一般实验室应届生毕业每年去的公司都差不多，要善于与实验室毕业的前辈联系，询问他们的建议，他们一般也会毫无保留地给予非常善意的回答。

5. 忠言也顺耳

找工作过程中的磕磕碰碰让我头破血流、身心疲惫，但同时也受益匪浅。最大的教训就是应届生的水平一般不会差距太大，如果想把工作找好，就要下真工夫、下苦工夫，就跟高考一样，绝对是天道酬勤，水到渠成。

9月份第一波招聘会来的时候你就必须要把基础知识、算法、智力题、英语准备好了。否则你只能惨淡地接受教训并发奋努力在国庆节后第二波高潮之前加紧追赶了。不过这比较紧，效果往往不是太好。

最后，签约要慎重，如果觉得没有找到好工作，一定要坚持，不要以为后面没有机会了。进入招聘后半段，大多数公司都会补招，这是坚持到最后的人才有的机会。

2.5 书中自有编程法

涛哥，男，西安电子科技大学2012届硕士研究生，现就职于华为技术有限公司西安研究所。

我虽然找到了一份不错的工作，但很难说得上是成功的经验。这里总结求职过程中的经验教训，对后来的求职者应该还是有很大的警示作用与借鉴意义的。

1. 选择因人而异

经过近一个月的努力，最终我拿到了两个offer：一个是华为技术有限公司西安研究所的云计算研发的职位，另一个是腾讯深圳的无线终端开发的职位，最后我放弃了腾讯，选择了华为。之所以放弃薪水更可观的腾讯而选择华为，一方面是由于我做的项目都是Java语言开发的，自身对C++不太熟悉，腾讯给的offer是终端开发，而我不想做终端开发；另一方面是深圳的生活压力大，我不想在工作压力大的同时，生活压力也这么大。所以我在国庆前就签约华为了，之后也没有再找工作，而是开始做毕业设计相关的事情。

2. 有所不为才能有所作为

联发科来的时候在全西安进行招聘，没投简历的也可以参加笔试。由于联发科是最早来招聘的大公司，所以参加笔试的人特别多。笔试题目不难，考的都是一些基本的数据结构、操作系统、计算机组成原理和C语言的知识，有三四道《编程之美》上的算法题，做完后自我感觉良好，顺理成章地收到了后续的面试通知。后续一共经过两轮面试，第一轮面试一共

10 分钟，都没问技术，只涉及了项目；第二轮面试我的是两个部门经理，也没问技术，时间大概有 20 分钟，提了项目和性格方面的问题，最后他们当场说给我 offer，然而我拒绝了，因为我想做后台开发，而他们提供的职位是终端开发。联发科也是一家非常不错企业，我只是不想去做自己不喜欢的东西罢了。也许正是因为我的放弃可以成就另外一个人的成功。

3. 落花有意流水无情

我一直想进的是阿里系的公司，原因有两点：第一点是我觉得整个阿里系的公司技术和氛围比较好；第二点是我非常佩服马云这个人，感觉对自己今后的成长会比较好。

阿里云是 9 月中旬来的，在西北工业大学笔试，一个小时要做十几道算法题，因为太想进阿里云了，所以非常紧张，最后没有发挥好，笔试都没通过。所以，在此提醒以后找工作的师弟师妹，找工作心态一定要放平，相信自己，没有什么大不了的。后面的笔试面试我就非常淡定，但还是被百度和淘宝两家公司给淘汰了。百度是 3 个小时做 10 道左右的算法题，应该是我参加的所有笔试中题目技术含量最高的，也是最难的，感觉就三道题目自己比较肯定答对了。后来收到了百度的面试通知，一面的时候问了 3 个技术题，一道是数学题，一道是问 LRU 页面调度算法用程序怎么实现，还有一道是文件分布式存储方面的，感觉回答的不太好，果然一面就把我给刷了，因为后续一直没有收到二面的消息。而淘宝跟阿里云的笔试很像，时间也是一个小时，题量比较大，题目比较难，我笔试也没通过，淘宝和阿里云的失利对我打击挺大的，它们是最想进的公司。

国庆之后没再找工作，一是因为没信心了，还有就是通过对这几家互联网公司的应聘，我发现自己的实力确实不行，首先是基础不扎实，对专业课中的知识点仅仅是知道皮毛，理解根本不深，然后是算法太差了。

4. 书中自有编程法

在这里推荐几本对找工作和以后搞软件技术有帮助的书籍，不过大家还是要有一个自己喜欢的方向（数据挖掘、图像处理、搜索等）：

专业基础：《深入理解计算机系统》、《操作系统》、《数据结构》。

算法：《算法导论》、《编程之美》、《编程珠玑》、《编程珠玑 2》、《计算机程序设计艺术》系列，算法的提高还要平常多做些题，网上有很多。

C：《C 语言程序设计》、《C 陷阱与缺陷》、《C 专家编程》、《C 和指针》。

C++：《C++ 程序设计语言》、《Effective C++》。

Linux：《UNIX 环境高级编程》、《Linux 设备驱动程序》、《深入理解 Linux 内核》、《UNIX 网络编程卷 1》、《UNIX 网络编程卷 2》。

Java：《Java 编程思想》、《Java 虚拟机》、《Java 与模式》。

5. 充电与实践非常重要

如果立志于做软件研发工作，那么求职时最重要的还是技术实力，而实力的练就需要平时的积累，现在还在上本科的同学要抓紧时间了，不管是工作还是上研究生，都要坚持每天给自己充电，如果有机会读研究生，尽量选准一个自己喜欢的方向，把大量的时间放在上面，而且要跟老师、师兄师姐以及企业里的人多交流。对于搞技术的人而言，实践是非常重要的，除了实验室的项目，大家还可以参加一些竞赛，如果在技术含量比较高的竞赛（ACM、腾讯创新大赛、华为创新设计大赛、中兴捧月程序设计大赛、百度之星等）中拿过奖，对于找工作会很有帮助。所以，我的建议是，如果实验室有比较好的项目，那就做实验室的项目，如果没有，那就多参加一些竞赛。

2.6 笔试成绩好，不会被鄙视

小白，女，电子科技大学 2012 届硕士研究生，现就职于中国电子科技集团某研究所。

要说给学弟学妹们留点建议，我想从找工作前的一些方面说起。毕竟找工作也就短短几个月，真正决定自己应聘结果的是最初的一些准备。当然，找工作确实也是个运气活，但是运气并非我们所能掌控的，所以做些我们能够做到的事情才是重要的。

1. 知己方能百战不殆

首先我要说的是，自己一定想清楚自己要什么。有不少人读研究生，其实没有想清楚自己以后到底想要从事什么样的工作。和大部分人一样，我从一开始就是完全听老师的话，没有任何自己的规划与计划，老师让做什么就做什么，也不想自己为什么要做，怎么做更好，而老师不管的情况下，自己也恍恍惚惚不知如何。所以我建议大家从一开始，至少研一修完学分开始，想想自己想要怎样的工作。想去外企的，尽早做好英语的准备，毕竟英语好，对于进外企还是很加分的。其次在自己所在的专业，专业基础扎实，成绩优秀也是外企较为看重的。对于进私企或其他单位而言，当然也需要为以后从事的工作做较多的准备。就我的专业而言，因为我是从事算法研究的，找工作的时候就比较苦恼，现在大多数 IT 企业，无论是招聘硬件工程师还是招聘软件工程师，都看重求职者的编程能力，他们很少将重心放在算法研究上，而我研究的算法面比较窄，他们也并不是很了解，所以在招聘过程中还是挺吃亏的。鉴于此，希望大家还是尽量多完善自己这方面的能力，不要等到找工作时，才手忙脚乱地开始准备。当然很多时候，作为学生，我们没有选择的权利，研究方向都是导师指定的，必须要做一些科研方面的东西，这个时候就要自己合理地安排好自己的工作了。倘若想进研究所，算法方面的研究还是必要的，有比较出色的文章发表也是很不错的。倘若想读博，一门心思搞学术，才是硬道理。所以，大家尽早衡量好自己的性格兴趣，选好方向，有的放矢，绝对是有益无害的。

确定好了工作类型，下一个问题就是工作地点的问题了。在正式开始找工作之前，希望大家结合自己的实际情况，和父母好好商量一下，如果有男/女朋友，也可以和男/女朋友好好商量一下，自己也多做些思考。毕竟全国各地，公司岗位那么多，海投的效果始终是不好的，个人也没有那么多精力。确定好几个工作地点，有针对性地选择准备，才能事半功倍。

2. 笔试成绩好，不会被鄙视

正式找工作开始前，还是要看看相关的找工作的书籍。程序员笔试面试题目还是和做的实际项目不太一样，笔试面试一般侧重细节，更加注重基础知识的考核，所以进行这方面的准备还是很有必要的。

对自己感兴趣的公司来招聘之前，还是要做好提前准备工作，对公司的初步了解，往年笔试面试题目等。公司面试时喜欢问为什么选择我们公司之类的问题，若能回答的比较得体，印象分会不错。

3. 诚者，天之道也；思诚者，人之道也

关于面试，本着相互尊重的态度，应该穿戴整洁，必要的场合还应该穿正装。第一次参加面试，心理上可能会有些紧张，其实也就是刚开始一两次会这样，面试笔试多了，也就习惯了，就会平淡很多，所以开始拿一两个自己不是很想去的公司练练手也是可以的。

在面试期间，做到礼貌、大方，对于对方的问题，做短暂的认真思考后有条理地回答比较重要。同时我想强调一个问题，诚信还是很重要的，我自己就在这块差点栽了跟头。面试前听说面试官是宣讲会的主讲人，很在意有没有去听他的宣讲，所以当面试官问我是否有去听过他

的宣讲会时，多了个心眼儿（之前也了解了大概宣讲的内容），我就回答说去了。结果没想到他突然问我啥时候去的，我完全不记得宣讲时间了，只记得是下午，大概说了个时间，结果差了一个小时，幸好反应快，说中午出去办事，回来的匆忙，急急忙忙去听，具体时间不是很清楚了，所以在此做个反面教材，给大家做个警示，诚信诚恳，才能获得好的机会。

群面是一个面试中经常遇到的事情，像华为、华赛、腾讯产品一般都有群面。对于群面，网上的讲解很多，大家可以了解一下。我个人觉得，在群面中并不是说一定要保持中庸，不能多说话。不该说的时候不要乱说，该说的时候一定要当仁不让。同时，注意语气态度，很多人说开了，就一副唯我独尊的样子，不给其他人说话的机会，其实不是话太多的错，是心态没摆正的错。

4. offer 不在多，在于精

生活中的痛苦大多不是因为选择造成的，而是选项太多造成的，所以我个人认为 offer 不在于多，在于精，一两个保底，然后为自己最中意的 offer 再认真一搏，这些就够了。关于如何选择最后在手里的 offer，其实做好工作类型和地点的考虑后，基本也能够确定了。在同等条件下，不仅需要衡量基本工资、绩效、奖金、福利等诸多因素，还要考虑所在地的生活成本等。

我教研室的一个同学，从研二上学期开始，就认定了一家研究所，详细了解他们的研究方向，在做实验室项目的同时，也参与了该研究所的相关项目。虽然研究得不够深入，但也做到了基本的了解，通过参与到研究所的项目，不断弥补自己在该方面的欠缺，最后在找工作的时候，几乎是一击即中，时间精力都节省了，身心相当轻松愉悦，所以希望大家以此为榜样。

5. 谋事在人，成事在天

我找工作的经历说难也不难，说顺利也不算顺利。确定要回家并且进研究所，却发现大部分研究所不要女生，而我厚着脸皮勉强面试了一家并且耐心地等到了最后，幸好最后还是顺利签了。虽然没有面试几家单位，但是心里承受的压力也不小，所以对于自己想去的单位，一定要尽可能地表达自己强烈的意愿和真诚，天道酬勤，不轻易放弃，只要不是太差，最后一丝机会也应该尽力抓住。“谋事在人，成事在天”，做好自己应做的努力，也就无悔了。

应届生找工作，确实是一件大事，但其实也并没有想象中的那么重要。人的一生还有那么长，你现在的认知未必和以后相同，机遇和发展都是不定的因素，所以良好的心态绝对是至关重要的，一次的选择，并不能确定你的一生。人的一生取决于人一直以来的认真和坚持，对于不可控的运气问题，保持一个正确的态度，切记不可急躁。真心不愿意的也不要屈就，感觉比较满意的，能签就先签上。其实就我的经验而言，只要你有实力、有耐心，保持好的心态，一般都会有较为满意的结果的。

2.7 不要一厢情愿做公司的备胎

追风少侠，男，西安电子科技大学 2012 届硕士研究生，现就职于杭州支付宝网络技术有限公司。

以下是我的个人经验与教训，大公司的面经笔试我就不谈了，网上到处都是，我只是想说点求职过程中需要注意的地方。

1. 好学校不如好成绩

笔试成绩的好坏直接决定你在一个面试官心目中的初期印象，而且很多面试顺序都是按照笔试成绩的顺序排列的，成绩排名越往后，面试官越是看不中你，要是你再迟到，那入围的机

会就更加少之又少。淘宝的那次面试就把我安排在下午三点半进行，可是那天我参加了支付宝的笔试耽误了，赶到面试地点都已经将近六点半了，一轮面试刚开始就被面试官反问试卷及格了没，我说及格了，结果没答对，然后就一直处于她问我答的环节，而她问的范围很广也很细，面试大概一共持续了将近一个小时十分钟，尽管如此，可是结果还是被淘汰了。

2. 不要一厢情愿做公司的备胎

在和公司签署三方协议之前，千万不要在一个公司上吊死，每个HR都会说自己的公司有多好多好。所以在此奉劝大家最好能联系一下在这家公司里面的员工，大体知道个大概情况，不要一味地相信给你的口头 offer。

作为口头 offer，本身没有法律效力。作为应届毕业生，要尽量多找几家公司，拿到多一点 offer，这样比较保险，同时在与用人单位谈工资时，也有很大的资本可谈。所以不要一厢情愿地做公司的备胎，要让自己有好几个备胎，这样不会处于被动局面。

3. 钱多事好离“家”近

如果就业有总部与分部之分的话，最好选择去企业的总部，因为企业总部和分部差别很大。在企业总部，大部分的资源都会汇集在那，机会也多，能够很快得到支持和帮助，学习的机会也更大，个人空间也更大，而在分部就会有很大局限性，很多好公司都不会把核心业务放在分部，顶多设置一个办事处，晋升机会一般也少，接触核心的东西也少，对个人的成长当然也会不利。

4. 论持久战

找工作是个艰苦的拉锯战、体力战、消耗战，有时候拿到了一个 offer，还希望有更好的 offer，有时候拿到了多个 offer，还需要考虑一段时间，不断地比较 offer 或者等待更好的 offer 的出现。所以一定要做好持久战的准备，并非人人都是千里马，也并非每个面试官都是伯乐。找工作，希望在短时间内得到面试官的认可也不是一件容易的事情，所以就算被这些“伯乐”拒绝也是很正常不过的事情，“此处不留人自有留人处”，好公司有的是，而且很多都会来第二次招人，不用担心找不到工作。

5. 实习是捷径

好公司人人都想去，可是好公司招聘的人数有限，并非人人都能进好公司，必然有很大一部分人最终与好公司擦肩而过。能进入好公司除了运气，更多的还是依靠实力。对于实力有点欠缺的人来说，千万要抓住该公司的实习机会，能去一定要去，因为通过实习最终留下来的可能性非常大，很大一部分实习的人都可以留下来成为正式员工。而且和你一起竞争去实习的同学数量与实力要远远少于校招的时候，而且很有可能，校招的人数会锐减。不仅如此，光鲜的实习经历也会增加简历的分量，对于应聘其他企业也会大有益处。

6. 做研究还是做项目

若不考虑读博士，就不要把精力仅仅只放在做研究上面，多做点工程性的项目。当然不可否认研究生阶段做科研给我带来的思维上的锻炼，但是公司青睐的大多数还是工程技术性人才。

2.8 小结

成功不可复制，所以不要盲目照搬别人的成功，因为每个人都是唯一的，都是不一样的：性格、环境、能力、智商、情商、机遇、身份都不一样，但是“他山之石，可以攻玉”，成功的方法、失败的教训却可以借鉴，成功也变得有章可循，认识自我、创造自我、成就自我，最终一样能够站在前人的肩膀上，用自己勤劳的双手、聪明的头脑取得成功，开创自己的美好明天。

企业面试笔试攻略

第3章

当无数 IT 企业来临的时候，到底是应该广撒网还是应该集中优势兵力重点突击某一个或某几个，一直是困扰应届毕业生的问题。其实不仅是应届毕业生，对于有工作经验的人而言，也会为此问题烦恼。对于这个问题，“仁者见仁，智者见智”，但无论选择哪一种方法，在进行求职时，都需要了解自己应聘企业的招聘相关信息，找准“攻击点”，最终必将事半功倍，取得意想不到的效果。

本章以当前主流 IT 企业为对象，如互联网企业、网络设备提供商、外企、研究所、国企（包括电信运营商以及银行等）、创业型企业等，对其面试笔试进行一对一的强力分析，包括招聘流程、面试笔试内容、笔试面试真题、面试需要注意的事项以及推荐知识点学习等，拨开这些企业面试笔试的神秘面纱，将其最直观的一面展现给求职者，以帮助读者顺利求职。

3.1 互联网企业

互联网的发展以人类无法想象的速度进行着，正如十年前没有谁能够想到互联网会对今天人们的生活产生如此深远和巨大的影响一样，我们也很难想象未来十年互联网会是什么样子，但毋庸置疑，未来互联网的高速发展仍然不会停止，一系列新的技术，如云计算、物联网、移动互联网等，将会继续蓬勃发展，对人们的生活产生巨大的变革，促进人类社会的飞速发展。

而伴随着互联网的发展，也产生了一大批优秀的互联网企业，有做门户网站的、有做搜索的、有做网络安全的、有做网络游戏的、有做电子商务的，林林总总。互联网的发展，铸就了这些行业巨头，而反过来，它们的存在也极大地推动了整个互联网产业的发展。

互联网行业作为当前的高薪行业，动辄十几万，甚至几十万的年薪，让无数青年才俊、IT 英才投身其中。而作为求职者，要想敲开这些名企的大门，也并非一件非常容易的事情，需要做好很多准备事项，否则最终的结果只能是“落花有意随流水，流水无心恋落花”。

1. 招聘流程

随着全球经济的回暖，互联网企业的招聘规模也日趋扩大，很多互联网企业也由以前的零零散散招聘，变为现在的大肆招兵买马、扩军备战，动辄招聘上千人。所以，作为求职者，挑战虽然存在，但机会依然很多。

互联网的招聘一般从每年的 9 月份开始，一直持续到 11 月份，他们会选择国内一些计算机专业比较强的大学作为招聘点，如清华大学、中国科学技术大学、上海交通大学、东南大学、浙江大学、华南理工大学、西安电子科技大学、武汉大学、西安交通大学、哈尔滨工业大学等名牌高校。

互联网的招聘流程一般也比较严格，主要包括以下几个步骤：网上注册简历→宣讲会→筛选简历→笔试→专业面试一→专业面试二→HR（人力资源）面试→综合面试→最终录用。需要注意的是，由于企业每年的招聘信息都可能会有变动，所以求职者应该更多地关注一下企业的招聘流程，做到实时更新。

2. 面试笔试注意事项

互联网是一个更新迅速的行业，所以在求职互联网企业的过程中，保持一颗平常心，相信自己，同时自己平时要多积累，多看与自己专业、职业相关的东西，比如上一些比较专业的技术网站，拓展自己的知识面，从而开阔自己的视野。

对于互联网企业的面试而言，首先，好好准备面试。因为互联网企业一般都比较年轻，他们比较注意对求职者归属感的培养，所以在求职之前，求职者需要了解该企业的企业文化，了解自己应聘的职位，只有知己知彼才能百战百胜。同时，分析各大企业历年的笔试面试题，往往能够发现很多一模一样的题，而且很多问题都是反复地被问及，所以一定要对一些经常被问到的问题事先做好相关的准备。例如，个人优缺点、个人兴趣爱好、如何自我介绍等，做到有备无患。对于简历的内容一定要做到严谨、仔细、认真，面试官通常会针对简历或材料提出问题，所以简历里最好可以突出重点，吸引面试官的注意力，进而争取到比较大的发挥空间。同时，自己需要事先准备好几个最后提问的问题，一般面试官在最后会问求职者对公司有没有其他问题需要进一步了解的，最好可以问上一两个，一方面可以对公司了解深入一些，另一方面也可以表现求职者的积极态度。

其次，不要不懂装懂，尤其是互联网企业的招聘。技术型面试中，面试官个个都是身经百战的老手，他们也是从求职者过来的，对求职者的心态了如指掌，所以在他们面前，不会就是不会，不要抱着侥幸的心理以为可以糊弄过关，其实企业对应届毕业生在技术上的要求不会太高，掌握好基础知识就行了，弄虚作假的人是得不到企业青睐的。不该说的话绝对不要多说，尤其是人力资源类的面试，多说一句不合适的话往往就搞砸了整个面试。

最后，就是调整好心态，充满信心，保持淡定。看着那么多人匆忙的脚步以及面试前的那种压抑的气氛，人很容易紧张，其实大可不必，互联网企业的面试官一般都是很有修养的，应聘的目的不是为了让求职者出丑，而是为了最大限度地发掘人才。面试每个人被问到的问题差异都很大，技术类面试一般针对你简历或者其他面试材料来问，除了技术问题，也涉及一些工作能力的考查，如效率观念等。人力资源类的会问到学习成绩、性格、沟通能力等问题，具体也有很大的不同，但是问题数量不算太多。

除了需要注意常见的面试笔试技巧与细节外，还要针对互联网企业招聘的特点进行一些必要的准备，避免一些不应该的错误，主要有以下一些方面的内容需要注意：

(1) 互联网企业一般对求职者的在校成绩没有硬性要求，但是会把成绩当做一个重要的衡量标准，所以成绩好是一个很大的优势。对于专业技术一流，但成绩不够理想的人来说，一样可以进入互联网企业，而不应该还未出战就认输了。

(2) 由于企业实际业务需求以及岗位本身的发展机遇，企业可能需要对求职者的工作地点做出相应的安排，所以求职者需要在面试中与面试官进行及时的沟通与协调，因为有些业务的实际工作地点可能与招聘宣讲的不一样，所以一定要注意工作地点的问题。

(3) 一段知名企业的实习经历，可以为自己找工作增加非常重的砝码，尤其是当你要进入某一个互联网企业时，通过在企业实习实现留在企业工作的愿望一点都不难。例如，某些互联网企业会在每年四五月份进行实习生招聘，提前在应届毕业生中发掘并笼络人才，所以对于希望进入该企业的应届毕业生而言，此不失为一种捷径。而且即使未能通过它的实习生甄选，仍然可以继续申请应届生校园招聘，一般也不会受到任何影响。

(4) 在校期间有机会多参与该企业组织的各种活动，很多互联网企业都会提供一些科技竞赛平台，发掘科技人才，如一些互联网企业组织的创新设计大赛、程序设计大赛等。除此之外，很多互联网企业会在一些高校设置俱乐部。一般而言，创新设计大赛获奖者以及企业俱乐

部的主要负责人都有进入该互联网企业的“绿色通道”，相比其他求职者机会更多。

(5) 从事研发的程序员一般都比较随意，除非是销售或是其他特殊场合（如银行、外企等），在面试的过程中，一般都不用穿正装，否则成不了鹤立鸡群，就成了鸡立鹤群，引起笑话，影响面试结果。

(6) 在对参与过的项目进行介绍时，不能一味地按照事前准备好的模板照本宣科，而应该根据所申请的工作的性质，多说一些与自己申请的工作内容相近的东西。例如，如果是搜索类企业，就可以多提及一些与搜索有关的项目；如果是安全类企业，就可以多提及一些有关网络安全的项目。

(7) 有些在北京设置有岗位的互联网企业，很难解决当地户口。由于每个公司得到的指标数量都是由北京市政府调控的，而且户口指标越来越严格，很多互联网企业在解决户口上不能给予绝对承诺，但是会尽力争取，除非是企业极力挽留的人，其他人获得户口的难度还是挺大的。

(8) 很多互联网企业为求职者提供的 offer 并非都完全一样，所以一定要区分顶级 offer 与普通 offer。顶级 offer 是企业给予面试笔试非常优秀者的绿色通道。一般而言，拿到顶级 offer 的求职者各个方面都较普通 offer 好，如待遇、户口、发展前景等。所以求职者一定要擦亮自己的眼睛，能够拿到企业的顶级 offer 或是有资格与企业谈条件的时候，一定不要放过机会。

(9) 很多互联网企业，实行内推制，即通过内部员工可以推荐校友、师弟师妹、朋友等来此工作，如果被推荐的人最终被该企业录取了，推荐者也会获得该企业提供的“伯乐奖”，这对推荐者与被推荐者来说都是一个莫大的荣耀。所以如果有机会，一定要通过各种渠道“求内推”。

(10) 互联网企业的面试看起来有点随意，其实对每个人而言机会都是均等的，它会给予求职者足够的机会来证明自己的能力。无论是名牌高校的毕业生还是普通高校的毕业生，无论是本科生还是研究生，只要足够优秀，互联网企业都会给予机会。

(11) 不要被同一根绳子绊倒两次。因为面试有时候可能有跨度，每一轮面试的面试官都不一样，但同一个问题可能会被不同的面试官提问。所以有些问题在面试的时候没回答好，面试完后一定要仔细思考，以防在下次或是下一个公司面试中也遇到同样的情况。最好能够将自己的面试内容做好记录，面试完回去后好好想想。

(12) 学会纸上写程序。求职者学习计算机时，一般都是在计算机上敲代码，不习惯在纸上写程序，但是在进行技术面试的时候，一般都需要在纸上写代码，在纸上写代码一般容易出错，思路也比较紊乱，所以最好事先多练习。

(13) 建议准备一个日程本，记录每一次宣讲会、笔试和面试的时间，这样一旦企业打电话来预约面试，可以马上查找日程本上的空闲时间，不至于发生时间上的冲突。每投一份简历，记录下企业的职位和要求，如果一段时间以后（1 个月或更长）有面试机会，可以翻出来看看，有所准备。以免因为投递简历太多，最后造成面试笔试张冠李戴的后果。

(14) 互联网企业的行业特性导致在互联网企业工作，工作强度、工作压力都比较大，工作也比较辛苦，高薪意味着高付出，但高付出同时也会为自己带来高回报。高薪不是叫出来的，是踏踏实实干出来的。

(15) 大型互联网企业的用户群广泛，他们对海量数据处理很感兴趣，尤其是在面试笔试的过程中的压轴大题都是海量数据处理，所以在应聘前一定要研究海量数据的处理问题，做到有备无患。

3. 真题分析

以下摘选一些著名互联网企业的部分面试笔试真题以及考查知识点供读者参考。

- (1) extern 的作用。
- (2) strstr() 函数的作用。
- (3) Windows 下线程优先级问题。
- (4) 多方法交换 x 与 y 的值。
- (5) 指针的自加与引用。
- (6) 前置++与后置++。
- (7) inline 的作用。
- (8) 二维数组的表示。
- (9) ifndef 的作用。
- (10) KMP 算法。
- (11) 函数调用方式。
- (12) 重载函数。
- (13) 构造函数与析构函数。
- (14) 合并两个有序链表。
- (15) 逻辑推理—智力题。
- (16) 100 亿条记录的文本文件，取出重复数最多的前 10 条。
- (17) 设计一个双向链表，并提供一个可根据值删除元素的函数。
- (18) 二叉树的多种遍历算法实现。
- (19) 有读和写两个线程和一个队列，读线程从队列中读数据，写线程往队列中写数据。
- (20) stack, heap, memory-pool。
- (21) TCP 的流量控制和拥塞控制机制。
- (22) 写一个函数，返回一个字符串中只出现一次的第一个字符。
- (23) 求一个数组中第 k 大的数的位置。
- (24) 面向对象继承、多态问题，如多态的实现机制。
- (25) 内联函数什么时候不展开？
- (26) 成员初始化列表有什么作用？什么必须在成员初始化列表中进行初始化？
- (27) 指针与引用的区别。
- (28) 创建空类时，哪些成员函数是系统默认的？
- (29) 有 10w 个 IP 段，这些 IP 段之间都不重合，随便给定一个 IP，求出属于哪个 IP 段。
- (30) 网络编程（网络编程范式，非阻塞 connect）。
- (31) TCP/IP。
- (32) Linux 的命令、原理以及底层实现。
- (33) Linux 编程，包括所有互斥的方法、多线程编程、进程间通信。
- (34) 一个一维数轴上有不同的线段，求重复最长的两个线段。例如，a: 1~3, b: 2~7, c: 2~8, 最长重复是 b 和 c。
- (35) 有向带权图最短路径。
- (36) 内存溢出与内存泄露有什么区别？
- (37) 利用互斥量和条件变量设计一个消息队列，具有以下功能：1) 创建消息队列（消息中所含的元素）；2) 消息队列中插入消息；3) 取出一个消息（阻塞方式）；4) 取出第一消

息（非阻塞方式）。注意，互斥量、条件变量和队列由系统给定。

(38) 用非递归方法完成二叉树的遍历。

(39) `cnwap` 和 `cnnet` 的区别。

(40) 设计一个内存管理策略，要求可以保证多线程时的安全，防止内存越界等，效率不低于 `malloc()`/`free()` 函数。

(41) 排列组合问题。

(42) 若有序表的关键字序列为 (b, c, d, e, f, g, q, r, s, t)，则在二分查找关键字 b 的过程中，先后进行比较的关键字依次是什么？

(43) 有一个虚拟存储系统，若进程在内存中占 3 页（开始时内存为空），若采用先进出（FIFO）页面淘汰算法，当执行如下访问序列后，1,2,3,4,5,1,2,5,1,2,3,4,5，会发生多少缺页？

(44) 有一个顺序栈 S，元素 s1, s2, s3, s4, s5, s6 依次进栈，如果 6 个元素的出栈顺序为 s2, s3, s4, s6, s5, s1，则顺序栈的容量至少应该有多少？

(45) [0,2,1,4,3,9,5,8,6,7]是以数组形式存储的最小堆，删除堆顶元素 0 后的结果是多少？

(46) 某页式存储管理系统中，地址寄存器长度为 24 位，其中号占 14 位，则主存的分块大小是多少字节？

(47) 运算符重载。

(48) 各种排序算法的使用与比较。

(49) 一维数组默认初始化问题。

(50) `const char* p1 = "hello"; char* const p2 = "world";` 有什么区别？

(51) `struct` 与 `class` 有什么区别与联系？

(52) 函数指针与指针函数的区别。

(53) 指针数组与数组指针的区别。

(54) 大端小端。

(55) 虚函数问题。

(56) 如何判断单链表是否有环？

在互联网企业的面试中，除了一些常见的技术面试问题外，还有以下与项目、性格有关的问题：

(1) 自我介绍。

(2) 项目相关问题。

(3) 了解我们企业吗？

(4) 家乡是哪里的？为什么要来我们这座城市工作？

(5) 为什么会选择我们企业？

(6) 为什么选择这个职位？

(7) 竞赛获奖以及论文。

(8) 自己的职业规划是什么？

(9) 谈谈自己的优势与劣势。

(10) 你是怎么在团队合作中发挥作用的？

(11) 结合简历中的实习经历问一些细节。

(12) 对我们企业的理解，喜欢我们吗？

(13) 个人优缺点。

(14) 个人对薪资问题。

- (15) 可以实习吗?
- (16) 你的同学为什么不选择我们企业?
- (17) 如果你没有被录用, 你觉得可能是什么问题?
- (18) 你有什么问题要问吗?

4. 推荐知识点学习

通过真题发现, 知名的互联网企业一般考查的知识面比较广, 从基本的语言知识, 到面向对象技术, 从排序到二叉树, 从逻辑推理到海量数据处理, 从英语题到智力题, 都有涉及, 所以最好的准备是从平时积累开始, 拓宽自己的知识面。

同时由于互联网企业的侧重点往往不同, 针对这一特性, 需要重点突出某一点。例如, 如果是搜索为核心的互联网企业, 就需要更加侧重于算法、操作系统、数据库等相关知识的研究; 如果是电子商务企业, 则除了基础知识以外, 还需要学习一些 Java 相关知识; 如果是网络安全企业, 则还需要学习有关软件安全、网络安全的专业知识。

但总的来说, 重点还是应该放在学习 C/C++、数据结构与算法以及海量数据信息处理上。

3.2 网络设备提供商

互联网的巨大发展, 网络设备功不可没, 网络设备已经成为互联网发展的基石。伴随着 IT 业的发展, 现在很多网络设备提供商已经不再将目光只是锁定在这一块“蛋糕”上, 纷纷将触角伸展开来, 业务范围也变得越来越广泛: 云计算、智能手机、物联网, 正因为如此, 他们对人才的渴望仍旧极度迫切, 招聘规模也比较大, 待遇自然也比较给力, 所以很多优秀的毕业生都选择投身到了这样的企业。

1. 招聘流程

由于该类企业招聘规模一般比较大, 所以他们的校园招聘也比较早, 从每年的七八月份就开始了, 有的甚至在四五月份就开始校园宣讲, 他们的招聘流程一般为: 注册简历→笔试→技术面试→上机测试或性格测试或群面→主管面试。一般会在北京航空航天大学、西安电子科技大学、南开大学、武汉大学、湖南大学、北京邮电大学等高校举行校园宣讲会。

2. 面试笔试注意事项

在整个应聘过程中, 面试是最具有决定性意义的一个环节, 事关成败。同时, 面试也是求职者全面展示自身素质、能力、品质的最佳时机, 面试发挥出色, 可以弥补先前笔试或是其他条件, 如学历、专业上的一些不足。除了常见的面试注意事项外, 在该类企业的面试笔试过程中, 还应该注意以下几个方面的问题:

(1) 该类企业的招聘主要以综合素质考查和技术能力考查为主, 综合素质主要考查以下方面内容: 责任心、沟通能力、团队精神、主动性、学习新知识的能力、意愿等。通过招聘主要考查人员以下三个方面: 第一, 言谈举止、仪容、仪表; 第二, 心态(心理状态); 第三, 专业知识。

(2) 面试要低调, 待人诚恳, 题可以答不上, 但是一定要让面试官觉得你这个人踏实可靠。

(3) 第一轮面试一般是技术面, 只要态度够谦虚, 又参与过实际的项目研发, 一般都会给二面机会, 特别是需求量比较大的岗位, 诸如软件研发、云计算等。遇到会回答的问题时应该保持淡定, 遇到不会回答的问题时, 也要保持淡定, 该类企业通过技术面刷人并不多。该类企业的面试问题都是从提交的简历出发, 一点一点地问, 问题会一个比一个深入, 直到你回答不上或者面试官满意为止。

(4) 由于该类企业的规模比较庞大,在全国好多大城市都设置有研发基地,可能会根据岗位需要,对求职者进行岗位调整,有时会进行异地研发,所以求职者一定要做好心理准备。如果无法接受,一定要将自己的意愿表达清楚。

(5) 该类企业一般都有性格测试这个环节,性格测试反映求职者是否能够适应岗位要求,性格测试是刷人的一个重要环节。一般而言,在进行性格测试时,最好要能够保持前后题的一致性。同时,如果该类企业取消性格测试,很有可能会组织群面,群面也是该类企业刷人的一个重要环节,所以提前准备有关群面的技巧是非常有必要的。

(6) 该类企业在近几年开始增加了对研发类岗位的上机测试,用以考查求职者实际的编程能力。机试题目一般都非常简单,都是最常见的编程题,不涉及高深的算法,求职者可以选择 C/C++ 或 Java 语言进行编写源代码,所以在机试前认真仔细地在自己的计算机上多敲敲代码会有很好的帮助,否则在紧张的气氛里,很难发挥出自己的真实水平。

(7) 该类企业一般每年都会在五六月份组织一些在校学生参加的程序设计竞赛,获奖的学生除了获得奖品与证书外,一般还能享受到招聘“绿色通道”的优待。所以,如果有机会,在求职前,参加一下这些科技竞赛,对于个人水平的提高大有用处。

(8) 该类企业的面试一般很集中:技术面、群面、机试、性格测试、主管面试,几乎都被安排在一两天时间内完成,对人的体力与精力是一个极大的考验。

(9) 该类企业的待遇一般比较好,虽然相比互联网企业,可能相对低一些,但是该类企业对工作年限较长、业绩比较突出的优秀员工可能会提供股票、过渡房,所以总体福利也还不错。

(10) 在该类企业面试中,有时会有英语口语测试,对于求职者而言,能说尽量开口说。一定要明白一个道理,那就是说得不好是能力问题,不说就是态度问题了。

(11) 很多企业在与求职者签订协议的时候,都明确要求,求职者不允许未来跳槽到同类型的竞争企业中去。

3. 真题分析

某知名网络设备提供商技术类笔试题。

(1) 判断题(对的写 T, 错的写 F 并说明原因, 每小题 4 分, 共 20 分)。

- 1) 有数组定义 `inta[2][2]={ {1},{2,3}}`; 则 `a[0][1]` 的值为 0。()
- 2) `int(*ptr)()`, 则 `ptr` 是一维数组的名字。()
- 3) 指针在任何情况下都可进行 `>`, `<`, `>=`, `<=`, `=` 运算。()
- 4) `switch(c)` 语句中 `c` 可以是 `int, long, char, float, unsigned int` 类型。()
- 5) `#define print(x) printf("theno, "#x", is")`。()

(2) 填空题(共 30 分)。

1) 在 Windows 下, 写出运行结果, 每空 2 分, 共 10 分。

```
charstr[ ]="Hello";
char*p=str;
intn=10;
sizeof(str)=( )
sizeof(p)=( )
sizeof(n)=( )
voidfunc(char str[100])
{ }
sizeof(str)=( )
```

2) `voidsetmemory(char**p, intnum)`

```

    {*p=(char*)malloc(num);}
    voidtest(void)
    {char*str=NULL;
    getmemory(&str,100);
    strcpy(str,"hello");
    printf(str);
    }

```

运行 test() 函数有什么结果? () 10 分

3) 设 `intarr[]={6,7,8,9,10};`

```

    int*ptr=arr;
    (ptr++)+=123;
    printf("%d,%d",*ptr,*(++ptr));

```

程序输出为 () 10 分

(3) 编程题 (第一小题 20 分, 第二小题 30 分)。

1) 不使用库函数, 编写函数 `intstrcmp(char*source,char*dest)`, 相等返回 0, 不等返回-1。

2) 写一函数 `intfun(char*p)`, 判断一字符串是否为回文, 是返回 1, 不是返回 0, 出错返回-1。

面试中部分非技术问题。

- (1) 自我介绍。
- (2) 家乡是哪里的? 为什么选择留在这个城市?
- (3) 是否喜爱运动? 喜爱什么运动项目?
- (4) 性格如何? 内向、外向或中性?
- (5) 用英语进行简短的自我介绍。
- (6) 对于工作地点有什么要求? 是否能够服从公司的分配?
- (7) 项目有关。
- (8) 自己的优缺点。
- (9) 为什么要离职?
- (10) 说说你的个人发展计划。
- (11) 对软件外包的认识。
- (12) 对经常加班的态度。
- (13) 对长期出差的认识。
- (14) 对工作责任心、沟通能力、团队精神、主动性的认识。
- (15) 群面。
- (16) 性格测试。

面试中部分技术问题。

- (1) `struct` 与 `class` 的区别。
- (2) `error` 与 `exception` 的区别。
- (3) 常见的软件测试方法有哪些。
- (4) `int *const p`, `int const *p`, `int const *const p` 的区别。
- (5) 在字符串 `STR` 中找字符串 `substr` 的个数。
- (6) 将字符串右移 `N` 位。
- (7) 大端和小端的区别。
- (8) `strlen` 和 `sizeof` 的区别。

- (9) 指针与数组的区别。
- (10) C/C++如何读写文件。
- (11) 堆与栈的区别。
- (12) 虚函数与纯虚函数。
- (13) 程序在内存中如何分布。
- (14) 内存泄露。
- (15) 宏定义。
- (16) 静态全局变量与一般全局变量的区别，静态全局函数与一般全局函数的区别。
- (17) heap 与 stack 的区别。
- (18) 链表的后续遍历实现。
- (19) 有序单项链表的插入函数。
- (20) 根据简历上的项目提问。
- (21) 实时操作系统与非实时操作系统的区别。

某企业部分机试题。

(1) 求一个数组里面能被 3 整除的个数。给了题目框架，但框架不允许修改。

(2) 计算一个数组中的奇数值和偶数之和。

(3) 手机号码合法性判断，我国大陆运营商的手机号码标准格式为国家码+手机号码，如 8613888888888，其特点：长度为 13 位，以 86 的国家码打头，手机号码的每一位都是数字。请实现手机号码合法性判断的函数要求：如果手机号码合法，则返回 0；如果手机号码长度不合法，则返回 1；如果手机号码中包含非数字的字符，则返回 2；如果手机号码不是以 86 打头的，返回 3。

(4) 计算两个字符串中匹配项的字符串，并将匹配的字符串存储在 c[] 中。要求：字符串* 可以匹配任意一个字符串，直到下一个匹配字母为止，其中字符串 2 中允许有*；输出相匹配的字符串；只要一个字符不匹配，匹配过程就结束。例如，字符串 1 为 abcdefg，字符串 2 为 a*f，输出为 abcdef。

(5) 从两个数组的最后一个元素比较两个数组中不同元素的个数，如有 array1[5]= {77,21,1,3,5}, array2[3]={1,3,5}, 从 array1[4]与 array2[2]比较开始，到 array1[2]与 array[0]比较结束。这样得出它们不同的元素个数为 0，若 array1[6]={77,21,1,3,5,7}, 那么他们不同的元素为 3。函数原型为 int compare_array(int len1, int array1[], int len2, int array2[]);其中，len1 与 len2 分别为数组 array1[]和 array2[]的长度，函数返回值为两个数组不同元素的个数。

(6) 实现约瑟夫环。

(7) 有字符串表示的一个四则运算表达式，要求计算出该表达式的正确数值。四则运算即：加减乘除 “+ - * /”，另外该表达式中的数字只能是 1 位（数值范围 0~9）。若有不能整除的情况，按向下取整处理，如 8/3 得出值为 2。若有字符串 “8+7*2-9/3”，计算出其值为 19。主要考点：1) 字的字符形式变换为数字形式的方法；2) 数字的数字形式变换为数字的字符串形式的方法。

4. 推荐知识点学习

该类企业笔试涉及的知识面比较广、比较细，计算机系统、数据结构、面向对象编程、C/C++、软件工程、操作系统、数据库系统、计算机网络、无线通信无一不涉及，重点是 C/C++、数据结构与算法，而且对简历上的内容问的比较细。该类企业的招聘有时会包括群面与性格测试，而且一般都是通过这两个步骤刷人，所以应该在招聘前加强这两个方面知识的训

练。同时会有少量的英文口语交流，对于英语基础薄弱的求职者，最好能够做一些必要的准备工作。

3.3 外企

随着改革开放的不断进行，当中国向世界敞开胸怀，加入 WTO 的时候，无数外资企业抓住机会来到中国落地生根、开枝散叶，他们在带来精湛技术的同时，也带来了完善的管理模式，自然也受到了国人的青睐。相比其他类型的企业，外企薪酬待遇优厚，出国旅游、社会保险、年假、失业保险和住房公积金都比较齐全，而且外企在管理上，一般都有一套完善的规范，不存在本土企业自身的局限性。在这种模式下，员工的工作能力往往能够得到快速提高，所以进入外企工作，成了很多人的梦想。

1. 招聘流程

外企的招聘流程通常为：网申→笔试→技术面试一→技术面试二→直属部门经理面试→更高级别经理面试→HR 面试。

外企对人才的考核非常认真仔细，因为他们不愿意随意招聘到一个不适合的人，然后还得花大气力来培养，而愿意在人才的发掘上花大力气，大投入也在所不惜，所以外企的招聘流程看似复杂繁琐。在笔试题目上，他们费尽心思，出的题目都很有技术含量，能相对客观地反映出求职者的专业技能、英语水平、智力、表达能力等；在面试这个问题上，他们做的也同样很优秀，外企的面试少则两三轮，多则五六轮、七八轮，不仅考查求职者的专业技能，还会通过各种面试官的面试，来考查求职者的综合素养，所以整个求职过程需要的时间短则半个月到一个月，长则三个月，有时甚至半年。

不同的外企校招时间各不相同，主要是根据企业自身的情况来设定，所以在招聘季来临时，提前做好各方面准备非常重要。

2. 面试笔试注意事项

外企的招聘过程不同于其他类型的企业，所以在进行面试笔试的时候需要给予“特别照顾”。一般需要注意以下事项：

(1) 注意仪表。在外企面试时一定要穿比较正式的职业装，男生应穿西装，女生应穿套装，但不一定是名牌，外企不会看重这些，真正看重的是内在素养。同时，女生最好不要携带一些晶光闪闪、叮叮当当的饰物，也不要化浓妆或穿太时髦、太暴露的服饰，最好化淡妆，发型简单整洁，给人精明强干的感觉。

(2) 注意礼仪。不要嚼口香糖或抽烟，平常有这种习惯的到时要忍着点。喝水最忌讳的有两点：一是喝水出声，二是把水杯弄洒。一定要小心，把水杯放远一点，喝不喝都没关系；记住打喷嚏之前或之后一定要对面试官说 Excuse me；当面试结束时，不要忘记向面试官表达希望能够被录用的强烈愿望，在握手告辞之前，也可以问一句招聘的下一步内容是什么。

(3) 切忌谈论政治。在外企招聘中，一般不要涉及与政治有关的内容，即使谈到也要注意主观感情色彩不要太浓。

(4) 在外企的初次面试中，除非能确认面试官对你很感兴趣，否则不应该询问有关薪水、假期、奖金、退休等问题。当然，如果面试官询问你希望的薪水时，应表示你对工作机会的兴趣要大于对具体的薪水，此时可以说明你的期望薪水。

(5) 谈吐要清晰，尽量少用语气词。在外企的面试中，使用太多如“呢、啦、吧”等语气词或口头禅会把面试官弄得心烦意乱。语气词或口头禅太多会让面试官误以为求职者自信心

和准备不足，从而影响求职结果。

(6) 不要过多解释或道歉。如果面试迟到了，一句抱歉就行了，或者加上真实的原因。不要编故事，以为可以蒙混过关，其实面试官都很精明，如果说谎，很快会被他们识破，而且事情往往是越抹越黑。

(7) 不要当面询问面试结果。一般在面试结束后，客气地对面试官说声谢谢就行了。有些求职者可能为了体现上进心，在面试结束时，会向面试官套近乎，询问面试官对自己感觉怎样，有什么需要改进的地方，这完全没有必要，因为当天不可能知道结果；问了还有可能适得其反，如果真的很想知道结果，可以在面试后 3~5 天，电话询问或是邮件询问。

(8) 不要谈论工资。一般而言，外企是不会在招聘会上说出具体薪水的，因为在这一面试阶段还没有到谈论薪水细节的地步。而且，外企也不太喜欢完全冲着工资去的人。对于求职者而言，最好不要主动提问薪水问题，如果想知道薪水，可以通过已毕业的师兄师姐了解他们所在行业的大致工资幅度。

(9) 在面试的过程中，不要请求面试官帮忙。即使面试官是自己的校友或者朋友，不要套近乎说“多谢您帮忙了”，这是很不专业的做法，说这句话不仅起不到正面的效果，还可能帮倒忙。

(10) 在回答问题时，一定要给出明确态度，不要模棱两可。例如，当面试官询问求职者性格是外向还是内向时，有些求职者可能会回答，“和朋友在一起时我比较外向，而在家时我比较内向”。这种回答，表面上看，两种性格都沾边，其实就等于没有回答。所以，在回答面试官问题的时候，一定要选择一个明确的方向，并给出理由作为支持。

(11) 外企的笔试面试题一般会有一些开放性问题，如“为什么你要选择计算机专业”之类的，大部分题目都没有固定答案，主要是考查生活经历和工作态度等方面是否和企业文化相契合，只要用英文表达流畅就可以了。

(12) 外企需要的人才应该胆大、心细、脸皮厚。胆大，不用怕；心细，认真，不拖沓；“脸皮厚”，技术功底不是最主要的，而“执着精神”是最关键的。

(13) 英文很重要。外企笔试题都是英文，而且会涉及企业内其他国家的工程师的面试环节，所以英语的准备非常重要。如果英语不是太强的话，尽量在考前一个月，多做英文题，如 GRE 的推理题，英文数据结构题等。

(14) 信面经，但不全信。外企的门槛一般比较高，网上的一些有关非常著名的外企的面经一般都是由一些强人写的，他们站在他们的角度看问题，一些小的细节障碍、重难点对他们而言，可能太“小儿科”了，所以他们可能都不提，轻松地就跨过去了，而对于水平差距比较大的人而言，可能就是这些小细节就决定了成败。所以，对于网上有关非常著名的外企的面经，要抱着消化吸收的心态来学习，而不能完全按照这些面经的思路走。

(15) 外企的笔试题目一般比较多，题量比较大，笔试时答题速度一定要快。如果不是强人，那么还是做好足够的心理准备，尽快做会做的，把会做的做全、做好就完美了，不太会的也适当写点。

(16) 在笔试前，尽量总结历年的考题，客观题必考，数据结构与算法设计能力需要培养。主观题不同，一般都是发挥题。还会考测试用例的题目，所以一定要多找些资料，归纳总结。编程类题目一般都有与树相关的数据结构，而且算法多样，所以一定要认真准备。

(17) 笔试面试前，一定要调整心态。战略上藐视它，战术上重视它，发挥出自己的真实水平，不要因为太在意、太认真而发挥失误，保持一颗平淡的心。

(18) 平时多练习数据结构与算法的题，发散思维，也可尝试脱离计算机，在纸上手写程

序, 积累纸上写程序的经验。

(19) 外企的待遇丰厚的同时, 也有自身的一些劣势, 主要包括: 首先, 工作压力会比较大; 其次, 职业发展会存在“天花板”问题, 而且失业率比较高, 尤其是老员工, 一旦上了年纪, 如果还未能上到一定级别, 很有可能会被炒鱿鱼, 当遇上经济不景气的时候, 失业的可能性更大; 最后, 在外企容易变成“螺丝钉”, 外企职责分明, 工作分得很细, 所以最终可能自己的能力只专属于本职工作, 而不能发展其他职位的工作能力, 对于未来跳槽, 会有一定的影响。

3. 真题分析

2008 年某知名咨询公司笔试真题。

(1) 123456789 的火车经过如下轨道从左边入口处移到右边出口处 (每车只能进临时轨道 M 一次) 按照从左向右的顺序, 下面的结果不可能是_____。

- A. 123876549 B. 321987654
C. 321456798 D. 789651234

(2) 如果 M 只能容纳 4 列车。上面选项应该选哪个_____。

(3) 3 3 8 8 用四则运算符如何得出 24。

(4) C#编程实现: 可变长有序数组的插入 (无重复数据结点)。

(5) 数 a 和 b, 如何空间消耗最小交换 a b 中的数。

(6) For the following description about OOP, which is right?

1. An object can inherit the feature of another object;
2. A sub class can contain additional attribute or behaviors.
3. Encapsulation is used to hide as much as possible about the inner working of the interface.
4. Encapsulation prevents the program from becoming independent.
5. Polymorphism allows the methods to have different signature but with the same name.

- A. 1, 2 B. 1, 4 C. 2, 3 D. 3, 5 E. 4, 5

(7) Function club is used to simulate guests in a club. With 0 guest initially and 50 as max occupancy, when guests beyond limitation, they need to wait outside; when some guests leave the waiting list will decrease. The function will print out the number of guests in the club and waiting outside. The function declaration as follows: void club(int x); positive x stands for guests arrived, negative x stands for guests left from within the club. For example, club (40) prints 40,0; and then club (20) prints 50,10; and then club (-5) prints 50,5; and then club (-30) prints 25,0; and then club (-30) prints N/A; since it is impossible input. To make sure this function works as defined, we have the following set of data to pass into the function and check the results.

- | | | |
|---------------------|---------------|--------------------|
| a. 60 | b. 20 50 -10 | c. 40 -30 |
| d. 60 -5 -10 -10 10 | e. 10 -20 | f. 30 10 10 10 -60 |
| g. 10 10 10 | h. 10 -10 10 | |
| A. a, d, e, g | B. c, d, f, g | C. a, c, d, h |
| D. b, d, g, h | E. c, d, e, f | |

4. 推荐知识点学习

在非技术问题上, 外企比较强调英文水平、学习能力、表达能力、团队合作能力、沟通能力, 技术上主要侧重数据结构与算法、C/C++基础知识等。需要特别强调的是, 数据结构与算法需要平时积累, 很难靠突击取得成效, 所以平时自己要多练习算法题。

3.4 国企

2008 年金融危机，当民企、外企都在困难中艰难前行，大幅度裁员、降薪时，国企却依然坚挺，几乎没有受到巨大的影响，反而继续保持发展态势。经过这次风暴后，越来越多的人开始意识到，尽管国企有其自身的局限性，但是国企仍然是一个非常不错的选择，国企在求职者心中的地位自然也大大提升。而且，随着国企改革的进一步深化，国企在人才引进上也逐步与市场接轨，人事制度的进一步完善使招聘人才的手段也日趋科学合理，所有这些都使得国企招聘变得炙手可热，成为求职者心中的“香饽饽”。

1. 招聘流程

由于国企自身的特点，使其在招聘时，面试风格和其他类型的企业有所不同，具有明显的国企特色。国企一般对应届毕业生比较感兴趣，他们的校招时间一般也比其他类型的企业晚，所以有志于进入国企工作的人一定要有心理准备，是铁定心思地忽略其他企业的存在、一直等到它们的到来，还是先选定一家企业保底然后继续等待更好的出现？

国企的招聘流程为：投递简历—人事面—主任面—录用 offer。

2. 面试笔试注意事项

国企的面试笔试一般比较保守，对求职者也并不特别苛刻，但同时求职者自由发挥的余地也被限制地相对比较狭窄。所以，对于国企，求职者只要认真准备，在面试笔试的过程中，不犯一些典型的低级错误，最后过关还是有很大希望的。当然，希望要变为现实，并非脑门一热、金口一开即可实现，还需要天时、地利与人和。天时地利一般都能具备，主要还是需要个人不断地努力，除了需要掌握基本的面试技巧外，同时还要清楚以下一些方面的内容：

(1) 区别各种聘用制度。国企中并非所有的人都是体制内的人，有的是事业编制，而有的是劳务派遣，还有的是合同工，各种聘用制度下的人的待遇、福利、晋升机会等都会存在很大的区别，如公积金、年终奖以及各种补贴等，所以在与国企签订三方协议时，一定要弄清楚聘用制度。

(2) 不用担心会在国企内埋没自己，只要有能力，在国企里还是有很大发展空间的。是金子终究可以发光的，尤其是在大型的国企，如银行、能源等单位，人事制度、管理都是非常完善的，对人才也是非常重视的，只要有实力，一样可以有所作为。

(3) 除了技术性特别强的职位，一般国企招聘笔试、面试都不会重点考查求职者的专业知识，他们更注重的是求职者是否具有较高的政治素质、是否具有踏实肯干的良好品质、是否遵纪守法、是否认同企业文化、是否具有共同的价值观、是否脚踏实地等。所以，国企一般对学生党员、学生干部比较感兴趣，因此在面试过程中，求职者一般需要主动地向面试官表现出自己这方面的优势，如果本身的政治素质过硬，就更容易让面试官对你另眼相看了。

(4) 国企一般来说不太喜欢面试中个性张扬的人，中规中矩、举止行为朴实的求职者更容易得到面试官的青睐。

(5) 多才多艺可为面试加分。许多求职者在应聘国企时并不因为专业能力出众脱颖而出，却是以一技之长而得到面试官的赞赏。例如，棋琴书画皆会、球技出众、爱好广泛等。

(6) 着装要正式。对于特定的岗位，有时一定要穿正装。一般来讲，保守一些的颜色更易被接受，深色西装、白衬衫、黑皮带、黑皮鞋都是商务着装的首选。毕业生在应聘银行的时候，需要特别注意形象，但也不用大过关注品牌，对于刚毕业的学生而言，气质才是真正能够吸引面试官眼球的地方。同时，应届毕业生最好不要用香水，国企更多的是看中应届毕业生的

可塑性和发展性，如果喷香水，则会给人社会化的感觉，反而不好。另外，男生千万不要染发或是做发型；女生也一定要打扮得体，不要太妖娆，不要佩戴过多的首饰。可以表现得有朝气，但不能表现的不成熟。

(7) 国企比较重视学历与学习背景，较高的学历不但可以在求职时占有优势，进去以后工资待遇也会有一点差别，但差别并不大。但学历越高，晋升空间一般越大，发展前景一般更好。

(8) 同等条件的外地人与本地人，国企一般更喜欢招收本地人。例如，某些运营商、银行在其所属省级、市级、县级的分公司都愿意招聘在外地求学的本地人充实自己的队伍，因为这些本地人更熟悉当地的方言、文化等，对于未来开拓市场更有优势，所以如果有意回家乡发展的求职者，不失为一个巨大的机会。

(9) 国企很少对求职者进行英语面试，但许多国企也特别在意求职者的“门面”问题，虽然不会对求职者的英语水平进行直接考核，但是还是希望求职者具备一定的英语水平。例如求职者是否拥有国家英语四级或六级证书，是否通过了托福或者 GRE 考试。

(10) 由于评判求职者个人能力强弱具有一定的主观性和偏差，国企一般比较看重求职者手中是否有证书：国家级奖学金（包括国家奖学金、国家励志奖学金等）、学校奖学金、社会奖学金等各类奖学金，ACM 竞赛获奖、数学建模获奖、高等数学竞赛等各类学科竞赛获奖，英语竞赛获奖、辩论赛获奖等，所以如果能够在简历或面试中，让面试官了解到自己手中的证书，对求职成功非常有用。

(11) 国企面试的问题常常会夹杂一些个人家庭背景等问题，比如是否是独生子女，父母的工作情况等，求职者只要如实回答就行，不用过多地揣摩其中所谓的“深意”，也不要撒谎，撒谎在哪里都是会被人厌恶的行为。

(12) 对于网上的面经，要取其精华、去其糟粕。在面试官这些“老国企”面前，不要畅谈对国企的认识，以免影响最终的面试结果。

3. 真题分析

2010 年某银行计算机类考试笔试题。

第 1 大题 判断题 20 道

第 2 大题 单项选择题 40 道

第 3 大题 简答题两道

(1) 构成死锁的必要条件是什么，如何检测死锁、解除死锁？

(2) 画出星形、树形、总线型、环形网络拓扑结构，并写出星形、总线型网络拓扑结构的特点。

第 4 大题目 综合应用题

(1) 多表查询：从 S（学号，姓名，年龄，生日）表和 SC（学号，课程号，成绩）中查询出没有选择课程号为 1001 的课程的所有学生的学号和姓名。

(2) 根据程序写出其输出结果。

```
void main( )
{
    static char arr[5]={'*', '*', '*', '*', '*'};
    int i,j,k;
    for(i = 0; i < 5; i++)
    {
        printf("\n");
        for(j = 0; j < i; j++) printf(" ");
```

```

        for(k = 0; k < 5; k++) printf("%c",arr[k]);
    }
}

```

(3) 写出以下程序实现的功能:

```

void main( )
{
    int a, b, c, *pa, *pb, *pc, *p;
    pa = &a; pb = &b; pc = &c;
    scanf("%d,%d,%d",pa,pb,pc);
    if(*pa > *pb) {*p=*pa;*pa=*pb;*pb=*p;}
    if(*pa > *pc) {*p=*pa;*pa=*pc;*pc=*p;}
    if(*pb > *pc) {*p=*pb;*pb=*pc;*pc=*p;}
    printf("%d,%d,%d",*pa,*pb,*pc);
}

```

(4) 写出表达式的后缀形式。

(5) 给出 A~H 8 个字母各自出现的概率, 写出它的最优二进制编码, 并画出最优二叉树和计算出平均码长。

面试技术题目。

(1) 需求分析中, 需要确定项目哪些方面的可行性?

(2) 是否熟悉 Java, Java 有什么特点?

(3) 构造函数与初始化列表的区别。

面试非技术真题。

(1) 自我介绍。

(2) 谈谈你的家庭情况。

(3) 你的业余爱好。

(4) 结合你的实际情况(教育、背景、经历、性格特点、优缺点、兴趣), 谈谈你对未来 3 年或 5 年的生活和工作的规划。

(5) 你如何看待一个人以往的工作经验和他今后工作绩效之间的关系?

(6) 请用英文陈述你对企业的认识和了解到的企业文化。

(7) 简历内容相关问题。

(8) 你为什么要选择我们?

(9) 你认为我们为什么要选你, 而不去选其他人?

(10) 你对我们的了解有多少?

(11) 你能为我们公司做些什么?

(12) 如何看待职业生涯中“骑驴找马”的现象?

(13) 周围的同学是怎样看你的?

(14) 面试了哪些公司?

(15) 是否签约了?

4. 推荐知识点学习

从考试内容上看, 国企的笔试对技术的要求一般都不是很高, 虽然囊括了计算机专业的所有课程: C/C++语言、面向对象、数据库、数据结构、操作系统、计算机组成原理、编译原理、多媒体技术、计算机网络、离散数学、设计模式等, 但考的都很基础。虽然知识面涉及的非常广, 但是也并非什么都考, 考试内容都是最常见的知识, 同时知识点并不是很深, 除此之外, 还涉及了少量的业务知识。

针对这些问题,要将计算机专业的知识好好认真复习,特别是常见的问题。另外,国企更多的是考查求职者的综合能力,包括为人处事能力、表达能力、学习能力、反应能力等。此外还需要提前准备一些企业文化的相关内容。

3.5 研究所

作为科研设计单位,近年来国家的投入也与日俱增,因此研究所员工的待遇、社会地位较之以往大幅提高,越来越多的应届毕业生都把进入研究所作为自己实现个人价值的途径。

1. 招聘流程

与民企、外企相比,研究所因为性质的不同,其招聘方式、招聘流程也不太相同。研究所的招聘一般都比较晚,很多都是在其他企业校园招聘完毕之后才开始进行招聘的,一般在10月底、11月初,有很多研究所甚至将招聘放到了第二年才进行。

一般研究所的招聘流程为:投递简历→人事面试→主任面试(集体面试)→录用 offer。研究所的宣讲招聘形式也很有特色,除了进行校园宣讲招聘外,还会组团到高校进行招聘。

2. 面试笔试注意事项

由于研究所不同于其他类型企业,虽然为科研设计单位,但也从事产品生产活动。随着事业单位的改革,也开始向企业转型。在面试笔试的时候,既保留了部分国企的形式风格,也具备现代企业的招聘形式。所以,在面试笔试研究所前,只有弄清楚与研究所相关的信息,才能更好地准备面试笔试,除了注意一些常规的面试技巧外,还需要注意以下一些事项:

(1) 研究所比较注重求职者的毕业学校、学历,越是效益好、职工福利好的研究所,要求求职者的毕业学校、学历越高,一般都要求双“211”或者双“985”的硕士毕业生(即本科与研究生所在学校都是“211”、“985”高校)。少量非核心专业可能要求会降低一些,但也会至少为“211”或“985”高校的本科。

(2) 研究所的待遇一般比企业要低,但是福利会好,基本工资不高,奖金占了收入的很大一块,除此之外,还有岗位工资、年终奖、住房补贴、交通补贴、餐补、单身宿舍、食堂等,而且研究所的住房公积金的比例也非常高。同时,由于研究所属于国家科研设计单位,对于新入职的员工一般有一笔安家费,而其他类型企业可能没有。

(3) 研究所一般没有笔试,即使有少许研究所有笔试,也只会侧重计算机基础知识的考核,而且笔试题目涉及的范围比较广,但不会太深,都是最常见的问题。研究所的面试也与其他类型企业的面试不同,研究所组织的面试一般分为两轮,第一轮是由人力资源部组织的人力资源面试,主要是对求职者的简历有一个基本的了解与确认,如家庭、学历、成绩、奖励等内容。通过第一轮面试的求职者一般能够进入到第二轮面试中,第二轮面试为部门主管面试,一般都是各个部门的主管进行多对一的集中面试。

(4) 在面试的过程中,研究所不会特别关注于知识细节,对于专业知识,他们更加希望专业对口、项目对口,除此之外,因为研究所有一定的国企特色,所以更加注重求职者的综合实力,包括语言表达能力、运动能力、个人才艺、家庭所在地、婚否等。

(5) 在与 HR 谈待遇的时候,注意区别事业编制与企业编制,是体制内的事业编制人员还是体制外的合同工。事业编制的好处是不用交养老保险,退休以后领取退休金,更加稳定。

(6) 研究所各个部门、科室、岗位待遇一般不一样,有时相差很大,所以作为刚入职的应届生,还是应该擦亮眼睛看清楚,除了了解整个研究所的效益以外,还要看部门的效益、发展前景等。

(7) 很多研究所在招聘的时候会说有福利分房的可能性, 作为应届毕业生, 也一定要自己心里清楚, 福利分房只针对体制内的人, 如果连事业编制都没有, 是没有分房可能性的。而且即使具有事业编制, 已经成为“体制内”的人了, 但如果要分房, 也是需要进行论资排辈的, 应届毕业生无论按什么进行排队都很难在短时间内分到房, 对于以后的分房, 谁都不准, 所以对分房子一定要持谨慎态度。

(8) 虽然研究所一般不自己培养本科生, 但是很多研究所都有硕士学位点或是博士学位点, 如果希望进某一个研究所, 可以通过保研或考研进入研究所, 毕业一般都可以留在研究所里工作。

(9) 研究所一般与高校的教师有一些项目往来, 如果有机会参与到这种项目中, 最后通过导师推荐或是项目匹配一般也能很容易地拿到研究所的 offer。需要注意的是, 大多数研究所的项目一般以底层开发为主, 即 C/C++, 对于上层的 Web 应用开发、云计算涉及的比较少, 所以以 Java、.NET 为强项的求职者可能会“吃亏”。在学习期间, 如果有机会的话, 尽量多参与一些以 C/C++ 为开发语言的嵌入式软件开发项目。

(10) 如果手头没有三方协议, 而研究所又急于签约时, 应届毕业生应该尝试与研究所签订一份双方认定的协议, 由于研究所毕竟是大单位, 所以不会欺骗求职者, 只要求职者是真心想去, 一般都会有机会。

(11) 国家对研究所有政策照顾。例如, 对于北京户口, 研究所一般可以解决, 而在其他类型企业是很难解决的, 对于想去北京安家的求职者而言, 进入研究所工作是个非常不错的选择。

(12) 早些年, 研究所一般只招收应届毕业生。近些年, 随着事业单位改革, 也会招收少量的非应届人员, 但数量也不多, 所以对于希望进研究所的人, 以应届毕业生的形式进入的可能性较之非应届机会更多一些。相反, 从事科研技术的人员, 如果在研究所工作若干年, 最后离职的, 一般也可以跳槽到高校、企业, 不会受到大的影响。

(13) 研究所的文化氛围较外企、民企更加浓厚, 对员工的人文关怀也更多, 工作环境也非常和谐。

(14) 外企的面试官希望招到一个成熟、活跃、聪明的新员工, 求职者最好个性突出、精力充沛、思想跳跃、富有创意。不同于外企, 研究所作为国家单位, 面试官更多地目光聚焦在对企业忠诚、政治上上进、遵守规章制度、乐于付出等方面。

(15) 随着国家对事业单位的改革, 很多研究所都在逐步向企业转型, 所以很多研究所都改名为公司名字或是成立了很多下属子公司, 所以求职者在求职的过程中, 一定不要因为研究所改名而错过了投递简历或是投递到了下属子公司而错过了研究所自身, 一定要及时地关注并弄清楚研究所的名称以及研究所与下属子公司招聘的相关情况, 以免出现理解错误造成出现“漏网之鱼”或张冠李戴的后果。

3. 真题分析

技术类笔试面试题很基础, 都是一些最常见的基础知识。以下是一些常见的研究所的非技术类面试题。

- (1) 自我介绍。
- (2) 你家是外地的, 为什么要留在这里工作?
- (3) 你有男/女朋友吗? 他们在哪里工作?
- (4) 做过什么项目?
- (5) 个人特长有哪些? 有什么兴趣爱好吗?

- (6) 你的优点是什么？你的缺点是什么？
- (7) 如果现在录用你，你能立刻来实习吗？
- (8) 高考成绩如何？重点线是多少？为什么不选择更好的学校或是就近读书？
- (9) 在学校期间，有什么事情是你觉得做得最好的？
- (10) 你的成绩不是很好，为什么没有取得非常好的成绩？
- (11) 除了学习以外，你还有别的兴趣爱好吗？
- (12) 你为什么愿意到我们这里来工作？
- (13) 有拿到其他企业的 offer 吗？为什么不去那里而要留我们所？

4. 推荐知识点学习

通过研究所的面试笔试真题不难发现，研究所一般不设置笔试，即使有也是程序设计基础最常考的内容。对于面试，涉及技术层面的也仅仅只是个人项目相关问题，所以在应聘研究所前，仔细研究个人项目，同时好好分析并研究该研究所做的主要项目，然后进行有针对性的准备。重点推荐本书软件工程章节知识以及常见的程序设计基础类知识的学习。

3.6 创业型企业

当拉里·佩奇、谢尔盖·布林在斯坦福大学宿舍里面研究搜索算法时，当马克·扎克伯格在哈佛大学为方便同学交流研究社交网络时，没有谁能够想得到，他们有一天能够改变整个世界。而最终他们做到了，只要有梦想、激情、能力、毅力，在这样一个开放的时代，创业不再是天方夜谭，成功也不再遥远。

社会的进步，离不开广大勤勤恳恳、脚踏实地辛勤工作的人，但真正推动社会进步的却是那些充满激情、将梦想变为现实的热血青年，“为自己打工”已经变得不再艰难，无数有志青年都选择了将自己的前途寄托在自己身上，进行了创业。创业，让生活更加充满梦想与挑战。

本节以目前国内最大的图片购物搜索引擎公司淘淘搜为例进行分析。

1. 招聘流程

淘淘搜总部设在杭州，并在北京设有运营团队。其中，研发人员 60% 以上，均在杭州。

淘淘搜的校园招聘一般启动于当年 9 月份，并于 10~11 月会进行全国巡回宣讲与招聘。每年招聘的人数约为 40 人，涉及的岗位包括算法研发工程师、C++ 开发工程师、前端开发工程师、Java 开发工程师、视觉设计师、产品助理、运营助理等。近些年，随着业务的不断扩大，招聘人数也在不断发展。

淘淘搜校招采取定点培养、优中选优的精英策略，定点高校包括浙江大学、华中科技大学、武汉大学、西安电子科技大学、四川大学、电子科技大学等。

淘淘搜的应聘流程一般比较严格，包括以下几个步骤：宣讲会→筛选简历→笔试→专业面试→HR 面试→综合面试→最终录用。而简历的筛选环节将被统一安排在相应的城市笔试环节开始一周进行，一般很少会通过简历进行剔除求职者，各个甄选环节的通过名单以及下一步安排一般都会通过淘淘搜 HR 的官方网站、短信、电话通知等形式予以传播，所以求职者需要及时关注相关信息。

2. 面试笔试注意事项

作为一家能够在 IT 浪潮中存活下来的创业型企业，淘淘搜一方面不断学习一些大企业成功的经验，同时也不断推陈出新，积极发扬互联网开放的精神。在求职创业型企业时，除了注

意常规的面试技巧和方法以外，还需要注意以下几个方面的内容：

(1) 在面试中，尽可能不要与面试官谈及该企业与其他大型企业比较的劣势，也不要过分关注眼前的利益，创业型企业短期内可能回报率不如某些成熟型的大企业。但是，如果在创业早期即成为企业骨干、核心，未来将会大有作为。

(2) 在面试前弄明白一个道理，在创业型企业里，每个员工所起的作用不仅仅局限在某一个单元或是某一个模块上，而是可能同时交叉进行多项工作。

(3) 创业型企业一般招聘规模都不是很大，所以精力比较有限，招聘所能涉及的城市以及大学有限，如果有志于进入创业型企业，最好能够提前做好必要的功课，如寻求内推机会等，否则很有可能会失去机会。

(4) 在面试的过程中，尽可能地在面试官面前体现出坚强、创新、团结的品质，因为创业型企业要在大企业间搏杀的夹缝中生存下来，靠得就是这样一群有梦想、有追求、充满激情，团结、博爱、创新、坚强的青年才俊。

(5) 在挑选创业型企业时，尽量挑选一些在沿海或是南方大城市的企业。相比较内地，沿海城市或是南方大城市经济更加发达，产业链更加齐全、市场更加开放，机会也更多，所以在此生根发芽的创业型企业，生命力更强，发展前景更好。

3. 真题分析

以下为淘淘搜 2011 年技术类笔试真题。

第一部分：基础知识

(1) 请用 C 或 Java 语言写出 BOOL 变量 flag 与“零值”比较的 if 语句_____。float 变量 x 与“零值”比较的 if 语句_____。char 指针变量 *p 与“零值”比较的 if 语句_____。

(2) 下面第_____个 for 循环是无限循环。

① for(int i=0; i==10; i+=0); ② for(int i=10; (i++^--i)==0; i+=0)

(3) C 语言参数的入栈顺序是_____。

(4) 用 C 语言预处理指令 #define 声明一个常数，用以表示一年有多少秒（假设一年有 365 天）_____。

(5) Linux 结束后台进程的命令是_____。

(6) 以下 Linux 命令对中，正确的是（ ）

① ls 和 sl; ② cat 和 tac; ③ more 和 erom; ④ exit 和 tixe

(7) 下面_____条命令是在 vi 编辑器中执行存盘退出的。

① :q; ② ZZ; ③ :q!; ④ :wq

(8) Linux 字符串查找命令是_____，nohup 命令的作用是_____。

(9) 在 OSI 7 层模型中，网络层的功能有_____。

① 确保数据的传送正确无误; ② 确定数据包如何转发与路由;
③ 在信道上传送比特流; ④ 纠错与流控。

简答：

(1) TCP 和 UDP 的区别是什么？

(2) 简单描述一下 TCP/IP 建立连接的过程。

(3) ping 命令是基于什么协议实现的，这个协议处于哪一层？

(4) 描述一下 Linux 的进程间通信方式。

(5) 继承、多态、封装、抽象，哪种面向对象的方法可以让你变得富有，为什么？

(6)《公孙龙子》记载：“齐王之谓尹文曰：‘寡人甚好士，以齐国无士，何也？’尹文曰：‘愿闻大王之所谓士者。’齐王无以应。”说明齐王_____。

① 昏庸无道；② 是个结巴；③ 不会下定义；④ 不会定义自己的需求。

(7) 蔺相如，司马相如；魏无忌，长孙无忌。下列哪一组对应关系与此类似，请做出解释。

① PHP, Python; ② JSP, servlet; ③ java, javascript; ④ C, C++。

解释：_____

第二部分：图像处理与分析

第三部分：C\C++程序设计与数据结构

(1) 请使用 C 语言给出下面变量 a 的定义。

a) An integer: _____

b) A pointer to an integer: _____

c) A pointer to a pointer to an integer: _____

d) An array of 10 integers: _____

e) An array of 10 pointers to integers: _____

f) A pointer to an array of 10 integers: _____

g) A pointer to a function that takes an integer as an argument and returns an integer:

h) An array of ten pointers to functions that take an integer argument and return an integer:

(2) 阅读以下 C++ 程序，写出运行结果。

```
class A
{
public:
void f1()
{
printf("A::f1\r\n");
}
virtual void f2()
{
printf("A::f2\r\n");
}
void callfunc()
{
printf("A::callfunc\r\n");
f1();
f2();
}
};
class B:public A
{
public:
void f1()
{
printf("B::f1\r\n");
}
void f2()
{
printf("B::f2\r\n");
}
```



```

    }
    void callfunc( )
    {
        printf("B::callfunc\r\n");
        f1();
        f2();
    }
};
int main( )
{
    B *pB=new B;
    pB->callfunc( );
    A *pA=pB;
    pA->callfunc( );
    return 0;
}

```

程序输出:

(3) 实现两个 $N \times N$ 矩阵的乘法, 矩阵由一维数组表示。设矩阵 A_{NN} 和 B_{NN} 分别表示如下。

$$A_{NN} = \begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix}, \quad B_{NN} = \begin{pmatrix} b_{11} & \cdots & b_{1N} \\ \vdots & \ddots & \vdots \\ b_{N1} & \cdots & b_{NN} \end{pmatrix}$$

(4) 请用 C/C++ 编程实现将整数 12345 转换成字符串 (要求: ①C 或 C++ 编程语言可任选一种; ②在 Windows 和 Linux 环境下都可以编译通过并输出正确结果)。

(5) 请用 C/C++ 编程实现单链表的逆置 (要求: ①C 或 C++ 编程语言可任选一种; ②在 Windows 和 Linux 环境下都可以编译通过并输出正确结果; ③撰写 gcc 工程管理文件 makefile) (请填写在答题纸上, 注明题号)。

第四部分: QA

(1) 一套完整的测试应该由哪些阶段组成?

(2) 请试着比较一下黑盒测试、白盒测试、单元测试、集成测试、系统测试、验收测试的区别与联系。

(3) 对于图 3-1 所示的程序流程图的程序流, 采用语句覆盖法设计测试案例, 至少需要设计几个测试案例, 并请简述设计策略。

(4) 为了验证程序是否实现单模块功能, 需要进行 (A), 为了验证单模块和其他模块按照规定方式工作, 需要进行 (B), 请简述理由。

(A) a. 单元测试 b. 集成测试 c. 确认测试
d. 功能测试

(B) a. 单元测试 b. 集成测试 c. 功能测试
d. 系统测试

(5) 后台, 一个文本框, 要求输入 10~40 个长度的任意格式的字符串, 要求输入的字符可以在前台正常显示, 请据此设计测试用例及数据, 以完整把握功能的正常使用, 并阐述设计方法和思想。

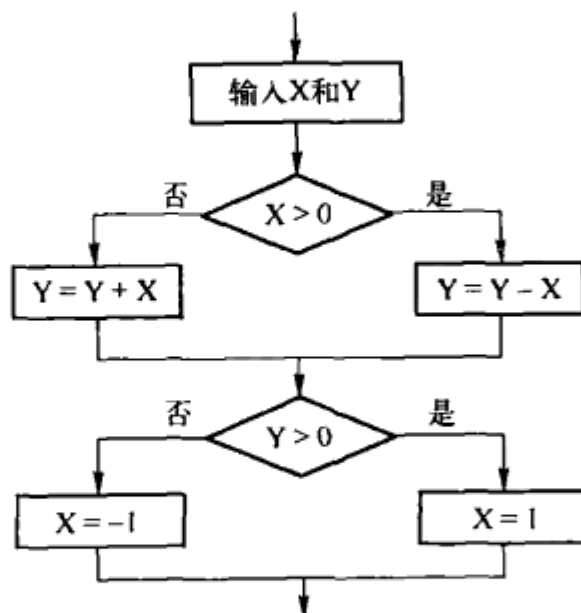


图 3-1 程序流程图

4. 推荐知识点学习

创业型公司对人才的考核与成熟型大型企业会不一样，在业务类岗位上，更偏重考查逻辑推理能力及创新能力，而技术类岗位则偏重考查专业知识的理解及自学能力，每个候选人可以在一套笔试题里选择自己感兴趣并擅长的题目。

所以，在准备创新型企业的面试笔试时，最好在夯实计算机基础专业知识的基础上，提高自身的综合能力。

3.7 如何抉择

求职就像择偶，好企业很多，素质很高的求职者也很多，但并非每个好员工都能成为每一个好的企业的“螺丝”，与每一个好企业实现无缝连接。所以，作为求职者，在选择一个企业前，一定要想清楚，这个企业适合自己吗？是自己希望的工作类型吗？自己能够在这家企业安心工作吗？否则，如果去了一家尽管很好，甚至是世界五百强的大企业，但是不适合自己的企业，就像谈恋爱没找到适合自己的对象一样，最终受伤的不仅是自己，还会伤害到别人。

不要以为自己很重要，对于企业而言，即使没有你的存在，企业的发展不会受到任何影响，而对于你自己而言，却可能付出惨痛的代价，失去青春、激情、甚至是前途。

一般情况下，对于求职者而言，在选择 offer 的时候需要综合考虑很多方面，以下列举出一些常见的项目供求职者参考。

(1) 是否因为感情问题，如爱情、亲情问题选择一座城市？自己的兴趣爱好是什么？是喜欢开放性程度好的城市还是开放性相对低一些的城市？

(2) 自己是否缺钱？互联网企业、外企一般待遇丰厚，对于缺钱的求职者而言，非常具有诱惑力。

(3) 自己是喜欢具有挑战性的工作还是喜欢相对稳定的工作？自己是新技术发烧友还是普通的技术控？一般来说，国企工作相对稳定，外企、民企的挑战性会更大一些。

(4) 自己是否对这个行业有着非常高的热情？虽然说干一行，爱一行，但更多的时候是爱一行了才能干好一行，兴趣是最好的老师。否则，在自己不喜欢的行业、企业，很有可能荒废自己，所以选择企业的时候一定要考虑清楚这个问题。

(5) 自己是否有志于创业？一般而言，在大公司磨炼若干年，积累一些大公司的经验，对个人成长与创业会有很大的帮助。

(6) 自己是否想过有一天跳槽到小企业当个小头目？在一些知名的大企业的工作经历，一般可以受到小企业的青睐。

(7) 自己是否是企业名人的粉丝？很多大企业的 CEO 或董事局主席都是行业的名人，在本行业有着很大的影响力，是很多青年才俊崇拜的偶像。

每个人求职的目的不同，想实现的目标也一样，有的人想去外企、有的人想去国企，有的人不怕加班就怕没钱、有的人希望轻松稳定而不在意钱多钱少，有的人希望把青春奉献给国家，有的人希望在外企大展身手。其实选择没有对错，求职也不是简单的买菜卖菜，更没有哪一家企业一定比另一家企业好。因此，最重要的是选择一个适合自己的企业，而不是为了攀比心、虚荣心，选择一个可能名号很响亮，却并不一定适合自己的企业。开开心心地工作才是最重要的，只有自己热爱自己的工作，才能把工作做好，进而在工作中实现人生价值，让自己的生活更加丰富多彩，否则企业的名头再响亮，提供的平台再好，可是自己不喜欢，一样很难在工作岗位上做出成绩来。

面试笔试技巧

第 4 章

不是看了本 C 语言编程书籍，就可以说精通 C 语言，会写一句 hello world，就可以自称程序员。程序员是一种职业，更是一种精神。只有经历了笔试面试的历练，求职者才能成为一名真正的程序员。而面试是用人单位经过精心设计，以交流、观察为手段，考查求职者是否具备其填报岗位所需的知识、技能、经验等的行为活动。它一般包括电话面试、群体面试、技术面试、HR 面试等多方面的内容。本章的主要内容为程序员笔试面试的技巧以及面试注意事项。

4.1 不打无准备之仗

虽然事先准备好的面试问题不一定会在面试中出现，但多做一些准备工作可以减少面试的失误，避免紧张，增强自信。面试中有些基本问题看似简单，其实要想圆润自如、有条不紊地回答好并不是想象中的那么容易，如自我介绍、薪水期望、为什么来应聘等。所以，在面试前进行适当的准备还是很有必要的，而且也会很有效果。

4.1.1 如何获取求职信息

求职成功与否，除了实力以外，还需要机会与运气。虽然说机会与运气是求职者无法预见和控制的，但是机会往往只会留给有准备的人，而获取最新、最快、最准的招聘信息对求职者而言，本身就是一种机会，很多求职成功的例子都是因为求职者及时准确地获取到了第一手招聘信息。“快人一步，快人一路”，赢在起跑线上，也许当众多竞争者还不知道招聘信息的时候，你已经早早地拿到了企业的 offer，所以企业招聘信息的获取对于求职者而言非常重要。

在获取招聘信息前，首先需要弄清楚为什么企业需要招聘人才。一个企业在发展或岗位人员调整的时候，就会产生相应的人员需求。对企业而言，招聘是获取人才的过程，也是一个成本投入的过程，为了控制成本并提高效率，企业采用的首先是内部人员的调剂、竞聘，其次采用的是员工推荐，在这些方式均不能达到目的的情况下，才会进行公开招聘。同时，随着 2008 年金融危机阴霾的消散，IT 企业也迎来了飞速发展的春天，云计算、移动互联网、Android 手机应用的风起云涌般的态势，使得企业对人才的需求也日益迫切，人才队伍建设已经成为各大 IT 企业发展的瓶颈。

在 IT 业这样一种大好形势下，对于计算机相关专业的求职者而言，找一份工作一点都没有难度，难在要找一份满意的工作。所以，为了获得一份适合自己的工作，求职者应做求职中的有心人，耳听八方，眼观六路，主动地发现需求，寻找机会，好的工作也许就在自己身边，找不到好工作的很大原因也许就是因为自己没有注意到这些信息。

获取招聘信息的方式有多种，具体如下：

第一种，通过公司招聘网站。该方法是最权威、最可信的，很多企业都有自己的网站以及人才招聘网，他们一般都设置有相应的投递简历的页面，会在第一时间将招聘职位、薪水待

遇、人才培养、需求数量等相关信息发布在该网站上。求职者可以通过访问企业招聘网站的方式获取招聘相关信息。这种方式也是最直观的方式。

第二种，通过人力资源服务网。现在有很多大型网站用于发布企业招聘信息，如智联招聘(<http://www.zhaopin.com>)等，所以在招聘高峰期来临前，趁早在智联招聘、中华英才网、大街网、前程无忧网等注册简历，一旦有招聘信息，能够在第一时间收到。

第三种，通过学校就业信息网。公司一般都会在学校就业信息网上发布招聘信息，所以求职者应该多关注各主要电子信息类大学的就业信息网，因为很多大公司在进行招聘的时候，受到很多因素的影响，很难走遍全国所有的大中城市，有时候只会去一些教育发达的城市进行宣讲或招聘，如北京、西安、杭州、上海、南京、成都、武汉等。同时，他们也会选择在以IT为强项的大学进行宣讲与招聘，如北京的清华大学、杭州的浙江大学、上海上海交通大学、西安的西安电子科技大学、南京的东南大学、武汉的华中科技大学、成都的电子科技大学等，所以关注这些大学的就业信息网往往能够有很大收获。

第四种，通过校园BBS。许多高校的BBS论坛区都设有专门的“招聘栏”以及工作讨论区，比较知名的BBS论坛有清华大学的水木社区、西安交通大学的兵马俑论坛、西安电子科技大学的西电好网等，一些学生、老师或其他人员会在这个版块上面发表一些较新的有关招聘信息的帖子，有关求职的文章供大家分享，还常常会在上面交谈他们的一些求职经验及心得等。通过浏览这些文章，也可以得到一些较新的招聘信息。

第五种，通过企业内部校友、实验室师兄姐妹。很多企业内部的员工对企业的招聘信息、企业文化、员工发展等都比较了解，能第一时间掌握企业的招聘相关信息，所以从他们口中一般可以获取很多招聘信息，有些甚至不为外人所熟知。

第六种，通过亲戚、朋友以及同学。“它山之石，可以攻玉”，社会是四通八达的，找工作也是，要学会利用自己的各种资源获取信息，做到多途径获得有用信息。

第七种，通过院系所办公室。每位学生都分属于各个学院、系、研究所等单位，而各个学院、系、研究所的专业、方向各不相同，有些企业可能只是针对个别院、系、研究所的，为了节省人力成本与精力，他们会进行小范围的笔试面试，所以许多用人单位比较习惯直接和各个教学单位联系用人事宜，尤其是用人规模比较小时，不用举办招聘会，一般都到办公室进行信息了解和招聘宣传。

第八种，通过校园俱乐部。很多大型IT企业越来越注重与大学的联系，会在所在高校以创建俱乐部的形式吸引优秀人才早日了解企业的项目以及文化，如微软创新俱乐部、腾讯创新俱乐部、华为创新俱乐部等，同时高校俱乐部也成为企业与校园的一个桥梁，很多时候都能更及时地收到一些招聘的消息。

第九种，通过其他媒介，如报纸、电视、广告牌等。有时候报纸也会有单独的招聘版块，电视上也会有少量的人才招聘计划节目。

总之，找工作最好是充分利用各种可用的信息资源，不放过任何可能的机会，信息是成败的关键。

4.1.2 如何制作一份受用人单位青睐的简历

简历是求职者给招聘单位的一份简要的书面自我介绍。它一般包含求职者的姓名、性别、出生年月、联系方式、籍贯、教育背景、联系方式、自我评价、项目经历、学习经历、荣誉与成就、求职愿望、对这份工作的简要理解等内容。需要注意的是，不要让平淡的事情冲淡了简历的吸引力，如果涉及过多的小事，会降低整个简历的质量水平，简历贵在精、简。

无论是通过面对面交流还是网络招聘，一份优秀的个人简历对于获得面试机会乃至最终取得心仪公司的 offer 都至关重要。简历的内容贵在真实，不要弄虚作假，抛开技术层面不讲，作假、说谎是一个人不诚信的表现，一个不诚信的人，任何公司都不会喜欢的。例如，项目经验里面，应该把自己参与做过的写上去，如果只是挂名而未实际参与的项目，最好别写上去，等到面试的时候，一旦被面试官识破，后果可想而知。而且最好不要心存侥幸，面试官们阅人无数，早已练就了火眼金睛，任何夸大和不实最终都会原形毕露。

简历要具有针对性，针对不同的岗位需求，简历的侧重点应不相同。技术性人才，往往简历上需要注重专业成绩以及项目经验；而市场型人才、管理型人才则除了上述两点外，还需要考查求职者是否在班级或学生会担任过干部等。针对不同性质的企业，也应该制定不一样的简历，而不应该“一种简历打天下”。对于应聘外企的求职者而言，一般除了中文简历外，最好还能携带英文简历以及推荐信；而对于应聘国企的求职者来说，就需要尽可能地将自己的成绩、荣誉证书等一并带上。

一份优秀简历与一般简历的区别见表 4-1。

表 4-1 优秀简历与一般简历的区别

| 项 目 | 普 通 版 | 优 秀 版 |
|-----------|-----------------------------|------------------------------------|
| 个人信息 | 全面。像户口调查表 | 简单。一般只包括最主要的信息，如地址、电话、E-mail 等 |
| 求职目标 | 一般没有 | 一般都有 |
| 获奖情况 | 一部分有、一部分没有。以罗列较多，没有归纳和分析 | 基本都有，除了描述以外还有对该奖项的归纳、分析 |
| 教育背景 | 加上很多课程名和奖励情况 | 加少量有针对性的专业课程名称，奖励单独一项进行介绍 |
| 个人特长 | 罗列较多，但是没有突出自己的独特之处 | 选择性很强，不随便写，具备一定水准了才写上去 |
| 工作经验/项目情况 | 较多，堆积一堆事情，没有轻重缓急，也不对其进行详细描述 | 有主次之分，最多不超过 3~4 项，每份工作或每个项目都能有详细描述 |
| 纸张 | 纸张过轻、不统一，五颜六色 | 白色纸张、80g 以上，比较讲究 |
| 页数 | 超过两页 | 1~2 页 |
| 性格特点、爱好 | 描述具体，而且很多 | 选择性添加、描述 |
| 排版 | 很差、不讲究 | 一丝不苟、十分讲究 |
| 直观印象 | 杂乱无章、无主次之分 | 精美舒畅、有轻有重 |
| 真实度 | 常常有虚假信息 | 真实不造假、只是艺术性地放大 |
| 文字 | 不规范，大小、字体不统一 | 规范、统一大小、统一字体 |
| 文字风格 | 平铺直叙、大段描述 | 言简意赅、分点描述 |

以下是一张中文简历的模板。

×××

手机：×××××××××××××× 邮箱：×××××××

联系地址：北京市×××××××大学×××××实验室（100000）

求职意向

研发工程师 R&D （2012 年 6 月毕业）

基本信息

姓名: ××× | 性别: × | 出生年月: ×××××× | 籍贯: 山东

教育背景

- ◆ **硕士:** 2009.9 至今 ××××××大学 计算机科学与技术
研究方向: ×××××× 成绩: 前 10%
- ◆ **本科:** 2005.9~2009.7 ××××××大学 软件工程
以软件工程专业**第二名**(共 333 人)的成绩保送至××××××大学, 成绩: 前 1%

荣誉奖励

- ◆ 2008.2 参加“国际大学生数学建模竞赛”, 获得**国际二等奖**。
- ◆ 2007.9 参加“全国大学生数学建模竞赛”, 获得**陕西三等奖**。
- ◆ 2006 年 5 月被评为“新生学习之星”, 并获得**校特等奖学金 (2/333)**; 2005~2006 年度被评为“优秀学生”, 并获得**国家奖学金 (5/333)**; 2006~2007 年度“优秀学习标兵”, 并获得**校特等奖学金 (4/333)**; 2007~2008 年度, 获得**校特等奖学金 (4/333)**。
- ◆ 2010.6 ×××大学研究生“公益活动贡献奖”。
- ◆ 2010.10 ××××××社区“义工之星”。
- ◆ 2011.6 ××××××研究生“公益活动贡献奖”。

项目经验

2011.8 至今 ××××××性能改进 ××××××公司系统平台组

- ◆ 项目描述: ××××××
- ◆ 所做工作: 项目主要负责人之一
- ◆ 相关技术: Hadoop, Java 语言
- ◆ 成果: 该项目正在进行中

2011.5~2011.8 ×××搜索 ××××××公司 搜索应用组

- ◆ 项目描述: ××××××
- ◆ 所做工作: ××××××
- ◆ 相关技术: ××××××
- ◆ 成果: 效果比较令人满意

专业技能

- ◆ 熟悉 **MapReduce 框架**, 尤其对 MapReduce 上的作业调度有一定的研究, 曾阅读并修改过 Hadoop 源代码。能在 Hadoop 上熟练使用 Java API、Hadoop Streaming 或者 Hadoop Pipes 进行分布式程序开发。
- ◆ 热爱**开源技术**, 熟悉常见的开源软件的使用方法和设计思想, 包括 Hadoop、Cassandra、Thrift、Scribe 等(这些均在实际项目中使用过)。
- ◆ 了解**数据库基础理论**, 对 MySQL、PostgreSQL 数据库有使用经验(曾修改 PostgreSQL 内核), 熟悉 TPC-H 测试基准。
- ◆ 熟悉 C/C++, 了解 Java。
- ◆ 熟悉 **Linux 环境**下程序开发(所有项目均在 Linux 环境下开发), 能够编写 socket、Shell Script 等, 曾利用 shell 脚本编写 Hadoop 批量作业测试脚本。
- ◆ 了解常用的**数据挖掘算法**, 曾在 Hadoop 上实现 Naive Bayes 分类算法和 Canopy 聚类算法。
- ◆ 英语六级, 能用英语书写专业论文。

社会活动

- ◆ 2010.4 组织并策划“走进心灵系列活动”之“邀请振华打工子弟学校学生及老师代表来我所参观”，该活动由“科技成果参观”、“主题讲座”等几个部分组成。
- ◆ 2010.1 参与组织××××大学元旦晚会，负责设计互动游戏和维护现场秩序等。
- ◆ 2009.3 担任一培训学校助教，负责改卷和讲解练习题等工作。

自我评价

- ◆ 踏实，努力，有上进心，有责任心。
- ◆ 乐于分享，喜欢帮助别人。
- ◆ 自学能力强，勤于知识梳理与总结。

对于大多数外企而言，一般在接收中文简历的同时，还需要接收一份英文简历，英文简历的编写内容基本与中文简历一致。值得注意的是，一定要通过英文简历，体现自己具备较高的英文水平。以下是一张英文简历模板。

×××
Tel: ×××-××××-×××× E-mail: ××××××
Address: Beijing (100190)

Job Objective

Software Engineer (R&D)

Education

- ◆ **Master:** Sept.2009~Jul.2012 ××× University,
Research fields: ×××××× GPA: top 10%
- ◆ **Bachelor:** Sept.2005~Jul.2009 ××× University
Software Engineering GPA: 3.8/4(top 1%)

Honors & Awards

- ◆ Feb.2008 Honorable Mention (Second prize) in American Mathematical Contest in Modeling
- ◆ Sep.2007 Third Prize for Shanxi Province, in China Undergraduates Mathematical Contest in Modeling
- ◆ Feb.2008 Excellent Students Scholarship Award, ××× University
- ◆ Feb.2007 Excellent Students Scholarship Award, ××× University
- ◆ Feb.2006 Excellent Students Scholarship Award, ××× University
- ◆ Jan.2006 National Scholarship, ××× University
- ◆ May.2005 Award of Top Star Student, ××× University

Project Experience

Aug.2011~Now **Improvement on ×××**

- ◆ Project Description:
- ◆ My work:
- ◆ Output: On-going

May.2011~Aug.2011 **Image search Engine v1.0**

- ◆ Project Description:

◆ My work:

◆ Output:

Professional Skills

- ◆ Expertise in **MapReduce Framework**, especially in MapReduce scheduler. Familiar with Hadoop programming with Java API, Hadoop Streaming or Hadoop pipes.
- ◆ Adore for **open-source software**, familiar with the usage and design principles of common open-source softwares like Hadoop, Cassandra, Thrift.
- ◆ Familiar with **databases, especially MySQL and PostgreSQL**.
- ◆ Have 3 years' experience of **C/C++ programming** and 1 year's experience of **Java programming**.
- ◆ Familiar with **Linux development environment** (all the above projects are under Fedora), and could write Socket and Shell Script, once wrote a batch shell to test Hadoop Schedulers.
- ◆ Know basic **data mining algorithm**, once develop Canopy and Naïve Bayes algorithm on Hadoop.
- ◆ CET 6, and good written English.

Extracurricular Activities

- ◆ Jun.2011 Award of public welfare activities, Institute of Computing Technology
- ◆ Oct.2010 Student Volunteer of HaiDian District, Beijing
- ◆ Jun. 2010 Award of public welfare activities, Institute of Computing Technology
- ◆ Jan.2010 Organizer of New Year's day party, Institute of Computing Technology
- ◆ Mar.2009 Teaching Assistant in a training school

Self Assessment

- ◆ Integrity and Diligent, with a strong sense of responsibility and good team-spirit
- ◆ Adore for sharing, considerate and thoughtful
- ◆ Have a strong ability of self-learning

一份高水平的简历模板是成功的关键，限于篇幅关系，读者也可以在相关的站点下载自己满意的简历模板。

4.1.3 如何高效地网申简历

近些年来，随着网络的普及，越来越多的人选择了通过网络投递简历（简称网投，也叫网络在线申请、网申）来进行求职，但同时越来越多的人发现一个问题：网投简历多数都是石沉大海，真正能够得到回复的并不多。其实，求职者投递的数十份甚至上百份简历之所以无法引起招聘方的注意，原因在于求职者投递简历的方式以及简历的内容。

网投简历首先要做到的是选对投递渠道。一般可以通过在招聘网站上直接单击“申请该职位”投递简历，也可以将自己的简历发送至招聘广告上公布的邮箱，一般建议如果在该网站已建立了最新的与该职位相匹配的简历，那么“申请该职位”通过该网站发送简历更好一些。因为通过这种方式人力资源（HR）能及时收到求职者的简历，而不会被当做垃圾邮件删除，而且对应聘的职位一目了然。

其次，不能盲目发送简历，也不能千篇一律，如果以为一份简历能够“打遍天下无敌手”，投什么职位都用同一份简历，那么就大错特错了，这种方式的成功率会非常低，但如果

为每一个职位都专门准备简历也是不切实际的，毕竟人的精力、时间有限，而求职过程本身就是一个耗费精力和时间的事情。考虑到不同企业、不同岗位都有各自的专业方向，侧重点会存在不同，所以简历内容不能一成不变，在投递不同公司以及不同职位的时候，需要对简历做适当修改，主要就是与岗位有关的相关信息要尽可能详述。

再次，在给用人单位发送简历的时候，应该尽量选择稳定性、可靠性高的私人邮箱，如果因为邮箱自身的缺陷，导致对方无法收到简历或者对方回复的信件丢失，那就太遗憾了。有很多邮箱会将一些群发邮件放入垃圾箱过滤掉，所以求职者在求职阶段，最好能够多关注一些被过滤掉的邮件。邮件的主题需要按照对方在招聘时（在职位广告中）已经声明的格式为填写，因为 HR 一天可能收到几百份甚至几千份简历，如果标题只写了“应聘”、“求职”，或是“简历”等，很难得到他们的关注，所以至少要写上应聘的职位这样才便于 HR 分门别类地去筛选。而且最好在标题中就写上自己的名字、联系方式、申请职位等相关信息，这样便于 HR 再次审核求职者的简历，并且在需要的时候及时给予回复。例如，有的用人单位就明确要求求职者在投递简历的时候，邮件的主题为：姓名（手机号）+学校+专业+求职岗位。

最后，申请的职位要准确，不能不写岗位，或是含糊其辞，应聘职位应该在企业给出的招聘范围内，不要自己随意发挥，创造一些新的职位名称，即使工作内容相似，但在职位名称方面一定要按照职位广告上所要求的填写，如招聘“渠道部总经理助理”，不要写成“总经理助理”或是“渠道助理”；招聘“副总裁秘书”，不要写成“总裁秘书”、“文秘”等。如果企业没有明确表态可以投递多个职位，切忌投寄多个职位，否则面试官会认为求职者对未来没有规划、信心不足以及专业性不强。

在进行网投的过程中，也有一些小技巧，比如在进行网投的同时附带求职信，表达对该企业有着浓厚的兴趣，然后简单地介绍自己的学历背景与工作经验，这样可以让 HR 在浏览简历时，能够立刻了解你，同时也突出了自己的优势。发送时间最好选择早上 8 点或下午 1 点，以保证简历能够在 HR 的邮箱里面排名靠前，因为 HR 一般会在上午 9 点半左右以及下午 2 点左右打开邮箱，在上午 11 点、下午 3 点左右通知你面试。不要用压缩包的形式发送简历，直接将简历内容写在电子邮件正文中而不是以附件的形式发送，当以附件发送的时候，会影响到 HR 的工作效率，面对海量的求职简历，很有可能被忽略了。

4.1.4 面试考查什么内容

面试是面试官与求职者就某一特定工作岗位以相互交流信息为目的、以判断求职者是否符合该职位的会谈过程。虽然用人单位无法通过一次面试准确、客观、清晰地把握其对工作的适合程度，但是却可以通过某些素质的测评进行综合有效地判断，从而得出能够基本符合事实的结论，所以面试仍然是一种重要的人才甄选方法。对于每个求职者，为了能够在众多的求职者中脱颖而出，首先需要弄清楚一个问题，就是面试官到底希望通过面试获取求职者的哪些信息。

在搞清楚面试考查的内容之前，首先看一下优秀程序员应该具备的素质，概括而言，主要有以下几个方面：

（1）仪表端庄。仪表端庄主要是针对求职者的外貌、气色、衣着、精神状态等方面的内容，任何行业对仪表风度都会有一定的要求，IT 业也不例外。在人们的思维定式中，相比较行为邋遢、随意的人，仪表端庄、衣着整洁、举止文明的人做事更有责任心，自我约束力更强。

（2）充满好奇心。IT 业高速发展，技术日新月异，而程序员必须对技术有着执着的追

求,勇于解开内心深处的迷惑和渴望,时刻充满好奇心对于一名优秀的程序员而言,是必备的素质之一。

(3) 逻辑缜密。计算机编程是一件需要严密逻辑和清晰思维的工作,有强大的数学或者科学背景的程序员通常更加容易取得成功。

(4) 快速阅读能力。程序员相当多的时间都用在阅读上,内容包括文档、其他人写的代码、API、注释等,有些程序员读得快,能很快理解,并且开始行动,另外一些程序员也许要多花三四倍、甚至更多的时间才能阅读完毕,这些程序员的工作效率肯定不如前者。

(5) 心细。没有程序不存在漏洞,虽然无法避免漏洞的发生,还是应该尽可能地减少漏洞的数量,所以对于程序员而言,关注细节,做事严谨,意味着他能够写出更高质量、更高效的代码,减少因为漏洞引起的损失。

(6) 工作态度认真。“态度决定细节,细节决定成败”,IT企业需要认真负责的人,所以优秀的程序员往往工作态度极其认真负责,遇事能够勇于担当,而不是逃避或者推诿。

(7) 快速学习能力。应用程序一般都与日常生活和企业运作相关联,如编写一个财务管理系统,就需要学习一些相关的财务知识,所以作为一名优秀的程序员,还应该具有快速学习新知识的能力。

(8) 自我学习能力。软件行业技术更新太快,新技术层出不穷,为了掌握更好的编程技能,好的程序员必须善于自我学习,不断给自己充电。

(9) 自我控制能力。IT企业往往会存在巨大的工作压力,当工作压力大或个人利益受到冲击时,优秀的程序员往往具备一定的韧性与耐力,能够克制、容忍、理智地对待,不致因情绪波动而影响工作。

(10) 激情与热情。任何工作都需要激情与热情,编程也不例外。IT界流传“611”或“711”代号来形容程序员工作辛苦。其实所谓“611”是指一周工作6天,每天工作11个小时,在如此高强度的工作节奏下,如果是以例行公事的态度工作,缺少对工作的激情与热情,那么这个人是很难做好这份工作的。

(11) 适应性强。程序员可能经常遇到短期的项目,需要变换不同的工作环境,与不同类型的人员打交道,所以优秀的程序员即使在客户公司工作,也能保持良好的工作状态,能够根据不同的情况,及时准确地应对,对于突发问题能够快速反应,对于意外事情能够处理得当。

(12) 沟通能力强。沟通能力强并不是指能够说一口流利的英语或普通话,而是指愿意沟通,善于沟通的一种能力。软件开发一般都需要团队协作完成,所以优秀的程序员要愿意并善于了解团队中其他人的想法,并且善于表达自己、倾听他人,既能较顺畅、准确地表达自己的思想、观点、意见、建议,同时也善于倾听他人的意见与建议。

(13) 服从管理。程序员是一个独立性很强的群体,有时可能会比较偏执,可能更加专注技术而忽视了市场和商业,但是从商业和其他角度来看,管理层不得不选择一个利益最大化的做法,此时程序员应该尊重领导的决定,而不是一意孤行。要知道,在一个团队中, $100-1=0$,而不是99。

虽然用人单位、用人者不同,具体的测评内容也有很大差别,但是总体而言,面试官就是希望通过面试这个环节,寻找出具备优秀程序员潜质的求职者,所以整个面试内容以及面试过程也都是围绕着这些方面展开的。鉴于此,在短暂的面试过程中,求职者要尽可能多地将自己以上这些方面的优点表现出来,不一定要全部做到,只要能够比其他求职者表现得更优秀、更有吸引力,就一定能够得到面试官的认可,从而在众多求职者中脱颖而出。

4.1.5 霸王面合适吗

霸王面也被称为“强面”，是求职过程中的一个名词，是指求职者在没有接到用人单位面试通知的情况下，直接到面试现场要求面试。

有的人觉得做霸王面破坏了企业的正常招聘流程，有失公平，对通过简历筛选或者通过笔试的人是一种不负责任的行为，而且霸王面一般也会给人留下不好的印象，感觉有点像死缠烂打，所以不可取。但也有人觉得霸王面可以给自己争取了一个机会，不管过程如何，只要最终结果满意就行了。霸王面是一个仁者见仁、智者见智的事情，好与不好都是相对的，但需要清楚的是，霸王面并不是求职成功的捷径，它与企业常规招聘计划有一定冲突，因此成功率一般很低。求职者如果没有一定的实力和特色，最好还是持谨慎态度。尤其不提倡那种对自身和企业特点都不了解的跟风霸王面，寄希望能出奇制胜，是不切实际也是不可取的。所以，一般不提倡霸王面这种做法，而且并非所有企业都能接受霸王面。

霸王面是一种非正常途径，不可经常如此。一般而言，以下 3 种情况的霸王面成功率比较高。1) 因为自己确实非常喜欢这家企业，但是确实未能通过简历筛选或是笔试，实属无奈之举。2) 自己因为其他事情（上课、实验室项目、与其他企业的笔试面试冲突等），错过了投递简历或是笔试环节，所以未能接到面试通知。3) 在简历筛选和笔试筛选的过程中，因为面试官筛选的疏忽导致被错过、被遗漏掉了，而并非自身条件达不到面试要求。否则，霸王面就是一种无赖、无理的行为。任何时候，都应该保持一个诚恳的态度。霸王面毕竟影响了正常流程，如果人人都效仿，单位肯定受不了，不然简历筛选和笔试就没意义了。

如果决定去参加霸王面，也不可贸然进行，需要从长计议。尽量制订一个周密的计划，没有计划的霸王面会给人鲁莽的感觉，本来霸王面就被定性为没有礼貌的行为，再加上这种鲁莽，是非常不利于求职成功的。在霸王面中，要体现自己的耐心和诚意，争取最后的成功。

4.1.6 非技术类笔试如何应答

评价一个人的能力，除了专业能力，还有一些非专业能力，如智力、沟通能力、反应能力等，所以在 IT 企业招聘过程的笔试环节中，并非所有的笔试内容都是 C/C++、数据结构与算法、操作系统等专业知识，也包括其他一些非技术类的知识，如智力题、推理题、作文题等。技术水平测试可以考查一个求职者的专业素养，而非技术类笔试则更加强调求职者的综合素质，包括数学分析能力、反应能力、临场应变能力、思维灵活性、文字表达能力、性格特征等内容。考查的形式多种多样，但与公务员考查相似，主要包括行测（占大多数）、性格测试（大部分都有）、应用文、开放问题等内容。

每个人都有自己的答题技巧，答题方式也各不相同，以下是一些相对比较好的答题技巧（以行测为例）：

（1）合理有效的时间管理。由于题目的难易不同，所以不要对所有题目都“绝对的公平”、都“一刀切”，要有轻重缓急，最好的做法是不按顺序回答。行测中有各种题型，如数量关系、图形推理、应用题、资料分析、文字逻辑等，而不同的人擅长的题型是不一样的，因此应该首先回答自己最擅长的问题。例如，如果对数字比较敏感，那么就先答数量关系。

（2）注意时间的把握。由于题量一般都比较大会比较大，可以先按照总时间/题数来计算每道题的平均答题时间，如 10 秒，如果看到某一道题 5 秒后还没思路，则马上放弃。在做行测题目的时候，以在最短的时间内拿到最多分为目标。

（3）平时多关注图表类题目，培养迅速抓住图表中的各个数字要素间相互的逻辑关系的

能力。

(4) 做题要集中精力，只有集中精力、全神贯注，才能将自己的水平最大限度地发挥出来。

(5) 学会关键字查找，通过关键字查找，能够提高做题效率。

(6) 提高估算能力，有很多时候，估算能够极大地提高做题速度，同时保证正确率。

除了行测以外，一些企业非常相信个人性格对入职匹配的影响，所以都会引入相关的性格测试题用于测试求职者的性格特性，看其是否适合所投递的职位。大多数情况下，只要按照自己的真实想法选择就行了，不要弄巧成拙，因为测试是为了得出正确的结果，因此大多测试题前后都有相互验证的题目。如果求职者自作聪明，选择该职位可能要求的性格选项，则很可能导致测试前后不符，这样很容易让企业发现你是个不诚实的人，从而首先予以筛除。

4.1.7 什么是职场暗语

随着求职大势的变迁发展，以往常规的面试套路，因为过于单调，已经被众多“面试达人”们挖掘出了各种“破解秘诀”，形成了类似“求职宝典”的各类“面经”。所以，面试官们也纷纷升级面试模式，为求职者们制作了更为隐蔽、间接的面试题目，让那些早已流传开来的“面试攻略”毫无用武之地，一些蕴涵丰富信息以更新面目出现的问话屡屡“秒杀”求职者，让求职者一头雾水。经常听到这样的疑问，“面试官从头到尾都表现出对我很感兴趣的样子，营造出马上就要录用我的氛围，为什么我最后还是被拒绝了？”“为什么 HR 会问我一些与专业、能力根本无关的怪问题，我感觉回答的也还行，为什么最后还是被拒绝了？”其实，这都是没有听懂面试“暗语”，没有听出面试官“弦外之音”的表现。“暗语”已经成为一种测试求职者心理素质、挖掘求职者内心真实想法的有效手段。理解这些面试中的暗语，对于求职者而言，非常重要。

以下是一些常见的面试暗语，求职者一定要弄清楚其中蕴含的深意。

(1) 请把简历先放在这，有消息我们会通知你的。

面试官说出这句话，则表明他对你已经“兴趣不大”了，为什么一定要等到有消息了再通知呢？难道现在不可以吗？所以，作为求职者，此时一定不要自作聪明、一厢情愿地等待着他们有消息通知你，因为一般不会有消息了。

(2) 我不是人力资源的，你别拘束，咱们就当是聊天，随便聊聊。

一般来说，能当面试官的人都是久经沙场的老将。所以，作为求职者应该时刻保持高度警觉，面试官不经意间问出来的问题，看似随意，很可能是他最想知道的。千万不要把面试过程当做聊天，当做朋友之间的侃大山，不要把面试官提出的问题当做是普通问题，而应该对每一个问题都仔细思考，认真回答，切忌不经过大脑的随意接话和回答。

(3) 是否可以谈谈你的要求和打算？

面试官在翻阅了求职者的简历后，说出这句话，很有可能是对求职者有兴趣，此时求职者应该尽量全方位地表现个人水平与才能，但也不能像王婆卖瓜那样引起对方的反感。

(4) 面试时只是“例行公事”式的问答。

如果面试时只是“例行公事”式的问答，没有什么激情或是主观性的赞许，此时希望就很渺茫了。但如果面试官对你的专长问得很细，而且表现出一种极大的关注与热情，那么此时希望会很大，作为求职者一定要抓住机会，将自己最好的一面展示在面试官面前。

(5) 你好，请坐！

简单的一句话，从面试官口中说出来其含义就大不同了。一般而言，面试官说出此话，

求职者回答“你好”或“您好”不重要，重要的是求职者是否“礼貌回应”和“坐不坐”。有的求职者的回应是“你好”或“您好”后直接落座，也有求职者回答“你好，谢谢”或“您好，谢谢”后落座，还有求职者一声不吭就坐下去，极个别求职者回答“谢谢”但不坐下来。前两种方法都可接受，后两者都不可接受。通过问候语，可以体现一个人的基本修养，直接影响面试官心目中的第一印象。

(6) 面试官向求职者探过身去。

在面试的过程中，面试官会有一些肢体语言，了解这些肢体语言对于了解面试官的心理情况以及面试的进展情况非常重要。例如，当面试官向求职者探过身去时，一般表明面试官对求职者很感兴趣；当面试官打呵欠或者目光呆滞、游移不定，甚至打开手机看时间或打电话、接电话时，一般表明面试官此时有了厌烦的情绪；而当面试官收拾文件或从椅子上站起来，一般表明此时面试官打算结束面试。针对面试官的肢体语言，求职者也应该迎合他们：当面试官很感兴趣时，应该继续陈述自己的观点；当面试官厌烦时，此时最好停下来，询问面试官是否愿意再继续听下去；当面试官打算结束面试，领会其用意，并准备好收场白，尽快地结束面试。

(7) 你从哪里知道我们的招聘信息的？

面试官提出这种问题，一方面是在评估招聘渠道的有效性，另一方面是想知道求职者是否有熟人介绍。一般而言，熟人介绍总体上会有加分，“不看僧面看佛面”，但是也不全是如此。如果是一个在单位里表现不佳的熟人介绍，则会起到相反的效果。而大多数面试官主要是为了评估自己企业发布招聘广告的有效性，顺带评估 HR 敬业与否。

(8) 你念书的时间还是比较富足的。

表面上看，这是对他人的高学历表示赞赏，但同时也是一语双关，如果“高学历”的同时还搭配上一个“高年龄”，就一定要提防面试官的质疑：比如有些人因为上学晚或是工作了以后再回来读的研究生，毕业年龄明显高出平均年龄。此时一定要向面试官解释清楚，否则面试官如果自己揣摩的话，往往会向不利于求职者的方向思考。例如，求职者年龄大的原因是高考复读过、考研用了两年甚至更长时间或者是先工作后读研等，如果面试官有了这种想法，最终的求职结果也就很难说了。

(9) 你有男/女朋友吗？你对异地恋爱怎么看待？

一般而言，面试官都会询问求职者的婚恋状况，一方面是对求职者个人问题的关心；另一方面，很有可能是试探你是否近期要结婚生子，将会给企业带来什么程度的负担。“能不能接受异地恋”，很有可能是考查你是否能够安心在一个地方工作，或者是暗示该岗位可能需要长期出差，试探求职者如何在感情和工作上作出抉择。与此类似的问题还有“是否生育？”、“小孩谁带？”。遇到这类问题，尽量要当场表态，避免将来的麻烦。

(10) 你还应聘过其他什么企业？

面试官提出这种问题是在考核你的职业生涯规划，同时顺便评估下你被其他企业录用或淘汰的可能性。当面试官对求职者提出此种问题时，表明面试官对求职者是基本肯定的，只是还不能下决定是否最终录用。如果你还应聘过其他企业，请最好选择相关联的岗位或行业回答。一般而言，如果应聘过其他企业，一定要说自己拿到了其他企业的 offer；如果其他的行业影响力高于现在面试的企业，无疑可以加大你自身的筹码，有时甚至可以因此拿到该企业的顶级 offer；如果行业影响力低于现在面试的企业，如果回答没有拿到 offer，则会给面试官一种误导：连这家企业都没有给你 offer，我们如果给你 offer 了，岂不是说明我们不如这家企业。

(11) 这是我的名片，你随时可以联系我。

在面试结束时，面试官起身将求职者送到门口，并主动与求职者握手，提供给求职者名片或是自己的个人电话，希望日后多加联系，此时求职者一定要明白，面试官已经对自己非常肯定了，这是被录用的前兆，因为很少有面试官会放下身段，对一个已经没有录用可能的求职者还如此“厚爱”。很多面试官在整个面试过程中会一直塑造出一种即将录用求职者的假象，如“你来到我们公司的话，有可能会比较忙”等模棱两可的表述，但如果面试官亲手将名片呈交，言谈中也流露出兴奋、积极的意向和表情，一般表明了一种接纳你的态度。

(12) 你担任职务很多，时间安排得过来吗？

对于有些职位，如销售等，学校的积极分子往往更具优势，但在应聘研发类岗位时，却并不一定吃香。面试官提出此类问题，其实就是对一些在学校当“领导”的学生的一种反感，大量的社交活动很有可能占据学业时间，从而导致专业基础不牢固。所以，针对上述问题，求职者在回答时，一定要告诉面试官，自己参与组织的“课外活动”并没有影响到自己的专业技能。

(13) 面试结束后，面试官说我们有消息会通知你的。

一般而言，面试官让求职者等通知，有多种可能性：没戏了；给你面试的人不是负责人，拿不了主意，还需要请示领导；公司对你不是特别满意，希望再多面试一些人，把你当做备胎，如果有比你更好的就不用你了，没有的话会找你；公司需要对面试完留下来的人进行重新选择，可能会安排二次面试。所以，当面试官说这句话时，表明此时成功的可能性不大，至少这一次不能给予肯定的回复，相反如果对方热情地和你握手言别，再加一句“欢迎你应聘本公司”，此时一般十有八九能和他做同事了。

(14) 我们会在几天后联系你。

一般而言，面试官说出这句话，表明了面试官对求职者还是很感兴趣的，尤其是当面试官仔细询问你所能接受的薪资情况等相关情况后，否则他们会尽快结束面谈，而不是多此一举。

(15) 面试官认为该结束面试时的暗语。

一般而言，求职者自我介绍之后，面试官会相应地提出各类问题，然后转向谈工作。面试官先会把工作、内容、职责介绍一番，接着让求职者谈谈今后工作的打算和设想，然后双方会谈及福利待遇问题，这些都是高潮话题，谈完之后你就应该主动作出告辞的姿态，不要盲目拖延时间。

面试官认为该结束面试时，往往会说以下暗示的话语来提醒求职者：

- 1) 我很感激你对我们公司这项工作的关注。
- 2) 真难为你了，跑了这么多路，多谢了。
- 3) 谢谢你对我们招聘工作的关心，我们一旦作出决定就会立即通知你。
- 4) 你的情况我们已经了解。你知道，在作出最后决定之前我们还要面试几位申请人。

此时，求职者应该主动站起身来，露出微笑，和面试官握手告辞，并且谢谢他，然后有礼貌地退出面试室。适时离场还包括不要在面试官结束谈话之前表现出浮躁不安、急欲离去或另去赴约的样子，过早地想离场会使面试官认为你应聘没有诚意或做事情没有耐心。

(16) 如果让你调到别的岗位，你愿意吗？

有些企业招收岗位和人员较多，在面试中，当听到面试官说出此话时，言外之意是该岗位也许已经“人满为患”或“名花有主”了，但企业对你兴趣不减，还是很希望你能成为企业的一员。面对这种提问，求职者应该迅速作出反应，如果认为对方是个不错的企业，你对新的

岗位又有一定的把握，也可以先进单位再选岗位；如果对方情况一般，新岗位又不太适合自己，最好当面回答不行。

(17) 你能来实习吗？

对于实习这种敏感的问题，面试官一般是不会轻易提及的，除非是确实对求职者很感兴趣，相中求职者了。当求职者遇到这种情况时，一定要清楚面试官的意图，他希望求职者表态了，如果确实可以去实习，一定及时地在面试官面前表达出来，这无疑可以给予自己更多的机会。

(18) 你什么时候能到岗？

当面试官问及到岗的时间时，表明面试官已经同意给 offer 了，此时只是为了确定求职者是否能够及时到岗并开始工作。如果确有难题千万不要遮遮掩掩，说清楚情况，诚实守信。

针对面试中存在的这种暗语，求职者在面试过程中，要多留一个心眼，多推敲推敲面试官的深意，仔细想想其中的“潜台词”。

4.1.8 如何克服面试中的紧张情绪

面试的成功与否，往小的方面讲，直接关系到求职者的工作问题，往大的方面讲，甚至事关求职者的前途命运。“男怕入错行，女怕嫁错郎”，在这种思想的熏陶下，人往往会产生巨大的心理负担，害怕出错，容易因此紧张、害怕，从而进入一种恶性循环，即“越紧张、越害怕，越害怕、越紧张”。

其实面试中紧张是一种非常常见的现象，在所难免，不紧张才不正常。紧张也并非坏事，适度的紧张，有利于刺激兴奋，但同样需要注意，过度的紧张会导致发挥失常。初次参加面试的人一般会因为紧张导致粗心大意、词不达意、结结巴巴，从而影响面试结果，所以克服面试紧张心理对于求职者非常重要。

克服紧张心理，首先需要保持一颗平和的心，大多数人在面对竞争的时候都会出现紧张。所以，当求职者觉得紧张的时候，想想其他求职者也会觉得紧张，它是一种客观存在的现象，而非特例。紧张既然无法回避，那就坦然面对，正视紧张。

其次，紧张往往是太在意的结果，与个人切身利益、前途命运的关联程度越强，则越紧张，其实相比较结果，更应该注重过程，“胜败乃兵家常事”。其实机会很多，大可不要觉得此次面试失败了就没有机会了，失败并不可怕，很多时候，失败的教训相比成功的经验甚至可以学到更多的知识，从而促使自己下一次更大的成功。

再次，紧张有时是因为害怕面试官造成的，大可不必这样，面试官也是人，他们也年轻过，也经历过求职中的磕磕碰碰，也犯过错误。尽管他们现阶段比你强，可能若干年前他们和你一样，所以不必害怕他们，也不必觉得自己不行，要放松。

最后，夯实基础，准备充分，从而提高自信心。在激烈竞争的职场中，一般需要具备信心、技能、沟通能力、创造能力与合作能力等多种技能，而信心又是最重要的，信心代表着一个人在事业中的精神状态以及对自己能力的正确认知。准备越充分，基础越扎实，自信心就会越强，成功率就会越高。

以下是几种消除过度紧张的技巧：

(1) 在面试前不要想着面试可能发生的事情，转移注意力。此时可以翻阅一本轻松有趣的杂志书籍，或者翻阅一下报纸，实在是没有心思，可以将随身携带的有关求职的书籍打开来翻翻，调整情绪，克服面试时的怯场心理。

(2) 注意控制谈话节奏。一般而言，紧张会导致语速加快，既不利于面试官听清讲话内

容，也会给人一种慌张的感觉，同时还容易出错，当然，讲话速度过慢，也会引起对方的反感，给人一种缺乏生气、沉闷的感觉。所以，讲话速度要适中，刚开始可以慢点以缓解自己的紧张情绪，随着心态的稳定，周围气氛的缓和，可以适当地加快语速。

(3) 回答面试官问题时，目光可以对准面试官的额头。目光游离不定的人，会给人一种不诚实、缺乏自信的印象，而两眼直盯着面试官，也会被误解为向对方挑战的意思，所以应该把目光集中在对方的额头上，既可以给对方以诚恳、自信的印象，表明自己善于倾听，也可以消除自己的紧张情绪。

作为求职者，即使最后被拒绝了，也不可因此而灰心丧气，一时的失误不等于面试失败，也不代表永远失败，重要的是求职过程。分析被拒绝的具体原因，总结经验教训，以新的姿态迎接下一次的面试才是最重要的。

4.1.9 面试礼仪有哪些

最能体现人修养的就是对礼仪的把握，礼仪作为中华民族的传统美德，已经成为评价一个素养高低的重要标准之一。不仅在面试中需要注重基本礼仪，在生活中更需要注重这些礼仪细节。具体而言，在面试时，需要注意以下几个方面：

(1) 时间观念。守时是做人的基本原则，也是职业道德的一个基本要求。当面试官与求职者商定好面试时间后，为了表示求职者的诚意以及对面试官的尊重，给面试官信任感，求职者尽量提前 10~15 分钟到达面试地点，提前到达的时间并非多余的，求职者可以利用这一时间熟悉一下周边环境，调整好自己的心态，并且作一些简单的准备。

面试时迟到或是匆匆忙忙赶到是致命的，如果面试迟到，那么不管有什么原因，都会被视为缺乏自我管理和约束能力，即缺乏职业能力，给面试官留下非常不好的印象。同时，面试迟到本身也是对别人的不尊重。由于大公司的面试时间安排紧凑，往往一次要安排很多人进行面试，如果迟到了几分钟，就很可能永远丧失了进入这家公司的机会，因为这是面试的第一道题，你的分值就被扣掉，后面也会因状态不佳而搞砸。所以如果路程较远，可以早点出门，但早到后不宜立刻进入办公室。

为了做到这一点，求职者一定要牢记面试的时间、地点等，如果条件允许的话，求职者最好能提前去一趟面试地点，了解乘车路线、所需时间等，以免因一时找不到地方或途中耽误时间而迟到。

(2) 注重礼仪。多使用“请”、“谢谢”、“您”、“非常荣幸”之类的话语，如果指定的面试官的房间大门关着，求职者应该首先敲门，在得到面试官的允许后方可进去。开关门动作要轻，以从容、自然为好。当与面试官见面时，要主动向面试官打招呼问好致意，称呼应当得体，一般称呼对方为老师。如果未征得面试官的同意，切勿急于落座，当面试官邀请你坐下时，首先应道声“谢谢”，方可入座。离开的时候应询问“还有什么要问的吗？”或“您需要我帮您叫下一个人的名字吗？”等内容，得到允许后应微笑起立，道谢并说“再见”，出门轻声关门。

和面试官见面时，应当自觉地将随身携带的有声物品（如手机、MP3）关掉或调成静音，在面试时电话响起是对面试官的不尊重以及对整个面试的不重视。

(3) 切忌小动作。面试时，要保持良好体态，当着面试官挖耳朵、擦鼻子、拉裙子、拨弄头发、打喷嚏、用力清喉咙等，应该极力避免。

在与面试官交流的过程中，需要注意以下禁忌事项：握手无力，与面试官见面握手时应当恰如其分轻轻一握；大大咧咧，左顾右盼，满不在乎；切忌坐立不安；双手总是不安稳、忙

个不停，做些玩弄领带、抚弄头发、掰关节、玩弄考官递过来的名片等动作；神经质般不停晃动；切忌弯腰弓背；眼神漂移不定或是死盯着面试官；面无表情，呆板；切忌手舞足蹈；言语离题。

总之，在面试的过程中，自始至终都要保持斯文有礼、不卑不亢，大方得体的言谈举止，谈吐谦虚谨慎，态度积极热情，不仅可大大提升求职者的形象，而且往往会使成功机会大增。

(4) 坐姿端正。在面试的过程中，求职者必须有良好的坐姿，以使面试中的沟通能够顺利进行。针对不同的坐具，也有一些不同的注意事项：如果是一张直背椅，切忌“瘫”在椅背上，背脊应该挺直，切勿弯腰弓背，也不要摇摆小腿；如果是一张沙发，则要尽量控制自己的身体，不要让身体陷坐下去。

其实，无论是坐硬椅子还是坐软沙发，都不应该太拘谨，应该保持轻松自如的坐势，双手最好平放在腿上，双眼平视面试官。面试过程中，最忌讳的坐姿就是“跷二郎腿”，应尽力避免。

(5) 回答谨慎。对于面试官提出的每一个问题，都要多加思考，认真回答。对方向你介绍情况时，要认真聆听，切忌在中途打断或抢问抢答，否则会给面试官急躁、鲁莽、不礼貌的印象。而且为了表示你已听懂面试官的话并对他的描述很感兴趣，可以在适当的时候点头或适当提问、答话。回答面试官的问题的时候，口齿要清晰，声音要适度，答话要简练、完整，注意控制语速。当面试官询问完毕后，对于听不懂的问题可要求其重复一遍，当不能回答某一问题时，应如实告诉面试官，含糊其辞和胡吹乱侃都会给人浮夸的感觉，最终导致面试失败。对重复的问题也要有耐心，不要表现出不耐烦。

只有懂得了这些基本的面试礼仪，才能给面试官一个好的印象，无论技术是好是坏，至少能够让面试官觉得自己是一个有教养的人。

4.1.10 面试需要准备什么内容

一般企业对求职者进行面试都会提前通知求职者，所以对于求职者，在面试前一定要做好相应的准备。

首先，求职者需要了解所要面试的企业的文化。在面试前，尽可能广泛地搜集与企业有关的资料和信息。文化不是一个实体，它是企业的精神，见证了企业的发展，预示着企业的将来，所以很多企业都非常注重自身的文化建设，注重对员工企业文化的培养。通过企业文化的了解，求职者结合自己的实际才能判断自己是否与企业的发展目标一致。

其次，要注意饮食卫生，晚上打点好装备，早点休息。面试的过程是一个心力交瘁的过程，除了对求职者的脑力进行考核外，奔波的疲劳更是一场体力的消耗，没有一个好的体力与精神状态，很难将自己的真实水平完全发挥出来，所以在面试前求职者应该尽量休息好，以饱满的热情迎接第二天的面试。

再次，携带好必需的材料，面试过程中可能面试官需要求职者随时拿出一些材料信息来核实求职者简历信息的真实性，所以求职者应该随身携带个人简历、个人成绩单、四六级证书、学位证书、毕业证书、获奖证书、专业资格任职证书、推荐信、已发表学术著作、学术论文原件以及复印件等资料，做到有备无患。同时，注意带上笔和本子，进行适当的记录与资料填写。

最后，着装整洁得体，避免穿着太鲜艳、太暴露的衣服。女性应尽量少佩戴金银首饰等，不要化妆太浓，切忌举止轻浮。

4.1.11 女生适合做程序员吗

程序员这个职业由于行业的特殊性，往往有一个不成文的规定就是“重男轻女”，更加青睐男性，其实也不只是在IT业，其他行业也存在这种现象，但尤其以IT业为甚。

虽然劳动法有专项条例保护女性，但在求职过程中的性别歧视却是个长期存在的问题，面试中经常会有女性求职者被问及一些尴尬问题。

作为程序员，可能经常需要出差，有时候需要半夜就起来去用户现场修改代码、调试程序，可能需要和同事一起熬夜加班，诸如此类的活动都是女程序员很难接受的，所以女生在进行面试的时候，总是会被面试官“刁难”，很难得到平等的机会以及相应的尊重。对于女程序员，社会上也存在着很大的偏见，以为女程序员只会编程，没有生活情趣，实则不然。

对于女程序员而言，最重要的是摆正心态。很多时候她们都不是很自信，她们会怀疑自己是否适合做技术，总觉得自己不如男性，也总在问自己的路在哪，其实大可不必这样杞人忧天，技术能力和性别无关，和个人有关，态度和努力是最重要的，任何以偏概全都是片面的。程序员的能力是多方面的，技术、合作、交流、态度等，任何一个都缺少不了，如果只是技术差点，那么提高的空间是很大的，多看看书，多实践，一点也不难。

企业在招聘女程序员的时候，经常考虑女性在家庭和婚姻中所承担的角色可能会影响到工作，所以面试时经常会提出许多相关的问题。因此，求职者能否回答好这些问题直接关系到求职能否成功。

以下是女生在求职的过程中经常会遇到的一些问题以及应对策略。

(1) 你觉得家庭和事业哪个更重要？

无论是男性还是女性，家庭与事业的矛盾都是同时存在的，只是受到传统思想的影响，即“男主外女主内”，认为女性应该更倾向于照顾家庭，导致在面试时女性不得不经常面对这样的提问。但是，也不是说企业就愿意听到女性求职者“工作至上，完全不顾家庭”的回答，对于企业来说，既希望你以事业为重，又希望你拥有一个幸福美满的家庭。

所以，在回答此类问题时，一般要表达出以下3个方面的意思：第一，家庭与事业之间确实存在矛盾，但并不是不能解决的；第二，无论是家庭还是事业，都可以体现出个人的价值；第三，当家庭和事业出现冲突时，自己有具体的处理方案，在大部分情况下，还是会以工作为重的。可以回答：我会结婚，会有自己的家庭，但我认为女人最重要的是能够保持自己的活力，工作对现代女性来说尤为重要。

(2) 婚后是否计划在近期内生育？

很多企业都会问女性生孩子的问题，其实这是一种干涉隐私和性别歧视。除了家庭与事业无法平衡外，企业提出此类问题，更多是出于成本的考虑。婚假和产假一般时间较长，在这段时间里，求职者无法给企业带来任何效益，还需要企业养着，作为以盈利为目的的企业，自然要考虑经济是否合算，对于企业而言，当然也无可厚非，但是这种做法确实很不规范，完全是一种歧视女性的表现。

但不管怎么样，结婚生子是每个女性必须正视的问题，所以没有必要掩饰什么，但回答一定要委婉，可以回答：我很重视自己的事业，因此我的决定以不影响我的工作和公司的利益为前提，谁都希望鱼和熊掌能够兼得，在一段时间内我会选择工作，因为拥有一份好的工作，将会为未来孩子的成长提供更为坚实的经济基础。我觉得总会有合适的时候让我两者兼得，我会理智地处理好这个问题的。

(3) 如果公司派你到外地出差，你的男朋友不同意你去，你会怎么办？

面对面试官提出的此类问题，为了获得他们的青睐，还是应该更加突出工作的重要性，可以参考以下几种回答：1) 公司安排我出差是工作上的需要，我和我的男朋友都是热爱工作和事业的人，相信我的男朋友会支持我的。如果他不同意，我也会说服他的。2) 我觉得如果公司派我出差，肯定有它的必要，所以我一定会去的，同时我也会了解到男朋友不同意的原因，然后想出一个让他不用担心的解决办法，决不让私人的事情影响到我工作中的事情。

对于行业内存在的性别歧视，女程序员首先要保持一颗平常心，要让自己的内心变得无比强大，碰到不信任的领导或男同事，要大胆地说出自己的想法，同时拿出有说服力的行动。不要轻言放弃，只要努力，一切皆有可能，女性相比男性，心更细，做事情更严谨，而程序员这个职业也需要严谨的人，测试行业也一般更加青睐女程序员，所以并不存在着女性不如男性的情况，只要踏踏实实，一样可以做得很好。

4.1.12 程序员是吃青春饭的吗

很多计算机相关专业的年轻人在进行择业的时候，不知道是受到什么因素的影响，对于程序员这个行业，都存在一个思想误区，就是觉得程序员的“职业生涯”很短暂，吃青春饭，等到年龄大了，如果不转行，就没用了，没有企业要了，生存都存在问题了，而且也举出了很多例证。例如，年长程序员需要的薪酬一般要比年轻程序员昂贵，相比之下年轻的程序员更能得到企业的青睐；年长程序员缺少灵活性，缺乏学习新知识的能力与动力，做事比较古板；年长程序员不愿意去干那些很辛苦的实际开发工作，编程水平一般，而只会指手画脚，往往给人眼高手低的感觉；年长程序员没有年轻程序员脑瓜灵活、思维敏捷等。一致认为程序员这个行业眼前虽然高薪，但却是以牺牲身体、时间换来的，还落下一身职业病，如颈椎病、腰椎间盘突出、高度近视等，所以这个行业没有什么前途，其实这是对程序员这个行业的一种误解。在美国，很多高薪的程序员，年纪都比较大，微软很多高手，都是四五十岁的人，而且都做的底层开发，但他们都成为了行业顶尖人物，之所以在中国很少见到比较年长的程序员，其主要原因不是因为年长的程序员都转行了，而是因为中国的信息技术起步较晚。在中国，信息技术大面积普及的时间大约是 1990 年以后，那个时候的年轻人，现在也只不过是四十岁。要说程序员辛苦，确实很辛苦，但毕竟“天上不会掉下馅饼”。说到职业病，也有点太过于片面，绝大多数行业都有职业病，而非仅仅是程序员行业。

其实，相比较年轻程序员，年长程序员更有竞争力。首先，年长程序员一般都有项目经验，之所以薪酬高昂，是有它的合理性与必要性的。年轻的程序员薪水一般比较少，他们一般没有经受过失败的教训，对于项目的认知与把握一般也不如年长的程序员，很多项目需要有激情的年轻人，同时也需要经验丰富的年长者进行架构、技术指导。其次，年长者的阅历较年轻的程序员更深、更广。很多深刻的见解并不是新人可以拥有的。由这些思想储备来提升的生产效率并不是可以用什么方法直接测量到的。最后，由于生理特性，年长者在智力方面确实不如年轻人了，年长的程序员比年轻的程序员的反应速度也相对慢一些，但正确的判断来自于经验，经验来自于常年的积累。

每个人都年轻过，也有年老的时候。有实力的程序员，无论是年轻还是年老，在哪里都受欢迎。所以程序员这个职业不仅不是吃“青春饭”的，而且这个职业还会让人永葆青春、充满活力。

4.1.13 为什么会被企业拒绝

求职的过程是求职者与面试官之间一个相互选择的过程，尽管求职者往往属于弱势的一

方，整个求职过程也称不上是绝对的公平，但就双向选择而言，相互之间都有接受或拒绝对方的权利，所以也可以称得上是相对公平。

一般面试官拒绝求职者有两种方法：一种是“明拒”，当场告诉求职者，不合格；另外一种方法是“暗拒”，即委婉的拒绝，当场不肯定也不否定，只是说让求职者先回去等消息，他们仔细商量后，会在一个星期之内给求职者消息，可是一个星期、两个星期、三个星期，甚至N个星期后仍然没有等到他们的消息。无论是哪一种拒绝方式，都是求职者不希望看到的，尤其是“暗拒”，虽然说面试官也是出于对求职者的尊重，但是作为求职者，最痛苦的事情不是没有希望，而是有了希望却最终破灭了，所以一家有素养的企业，即使是拒绝别人了也应该让对方及时知道。

被企业拒绝的原因不全是因为面试笔试的表现太糟糕，一般还有其他两个方面的原因：第一，求职者的能力暂时还没有达到企业需要的标准，无法胜任自己投递的岗位；第二，企业的工作暂时用不着求职者这样的人，求职者的能力已经超过了企业的需要，企业完全可以以更加低廉的价格找到适合的人。

针对以上两种原因，求职者一定要好好反思，做到“吃一堑，长一智”，在哪里跌倒就在哪里站起来，认真总结经验教训，而不是在被拒绝后一味地抱怨或是由此产生悲观情绪。

如果个人能力暂时不符合企业的职位要求，可能是自己对自己的期望太高了，此时应该考虑是否转向要求较低的初级职位。同时需要注意，在以后的学习工作中，一定要不断地提高自己的各方面的能力。求职者一定要全方位地了解自己的应聘企业，根据自己各个方面的情况以及所处的地域、目标行业、公司的综合情况，综合考虑，而不是盲目地高估自己的能力与水平。如果是个人的能力超过了职位的要求，企业无法提供一个匹配自己的薪水，此时要么委屈自己接受，要么果断放弃，在放弃的同时，保持一颗坚强和平静的心。如果是自己面试笔试表现糟糕，那么就需要注意了，有限的面试笔试时间，为什么没有博得面试官的好感，自己的失误之处在哪里？在哪里表现得不好让面试官产生了一种不好的印象？是因为自己说话不流畅导致面试官不理解，还是由于过于紧张导致问题回答不上来，或是因为自己不懂基本的礼仪等。

所以，当自己没有通过面试，被企业拒绝时，无论是因为哪一种原因，既不能一味地妄自菲薄，也不能毫不在意，要冷静地分析其中的原因，不断地反省，争取不断地提高，以求下一次能找到适合自己的工作。

4.1.14 如何准备集体面试

集体面试也被称为群面。计算机发展至今，软件开发已经不再是个人小作坊式的活动了，而是一个需要集思广益的团队合作过程，群面作为一个考查求职者团队合作能力的手段正越来越多地被应用于企业招聘中。

群面是企业常见的面试形式之一，它采用情景模拟的方式对考生进行集体面试。它一般将5~10人组织在一起，进行1个小时左右的与工作相关问题的讨论。在讨论过程中，每一个求职者地位平等，而且不指定领导，不指定求职者应坐的位置，让求职者自行安排组织，求职者需要通过自己的努力，争取到小组中公认的角色，并为小组讨论结果贡献自己的力量，在此过程中面试官通过观察，对求职者的组织协调能力、口头表达能力、分析问题能力、团队合作能力、辩论的说服能力、领导力、情绪控制能力等各方面的能力和素质是否达到拟任岗位的要求进行把握，从而确定是否求职者符合拟任岗位的需求。

群面的一般步骤如下：

(1) 接受问题，成员各自分别准备发言提纲。

(2) 小组成员轮流发言, 阐述自己的观点。

(3) 成员交叉讨论, 渐渐得出最佳方案。

(4) 解决方案总结并汇报讨论结果。

在群面中, 每个求职者给面试官最直接的印象就是风度、教养与见识, 在整个面试过程中, 面试官通过观察求职者发言的时机、发言的内容、何时停止、遭到反驳时的态度、倾听他人谈话时的态度等给予评价。

在群面中, 每个人都希望扮演一个能够被面试官接受也适合自己的角色, 可是往往事与愿违, 情况并不理想, 有的人精于辩论, 说话滔滔不绝, 不给别人说话的机会, 有的人一声不吭或者细声细语, 影响小组的最终表现, 这两种情况都不行。在群面中, 不是发言越多越好, 如果没有独到、深刻的观点, 发言太多相反会引起面试官的反感, 而如果不说或是小声说, 也很难引起面试官的注意, 同样得不到面试官的青睐。所以, 在整个群面的过程中, 掌握一定的技巧非常重要, 认真倾听他人观点、不紧不慢表现从容的发言者, 往往会获得较高的评价。

其实虽然没有明确的角色划分, 但是通常情况下, 可以粗略地将小组成员划分为以下几个角色:

(1) 领导。群面中, 领导起的作用很大, 作为一名优秀的管理者, 他应该根据小组其他求职者的专业和特长等, 合理恰当地进行分工, 并能把各阶段的陈述和总结机会, 合理恰当地分给小组的其他求职者。在团队中, 领导的思路非常重要, 只有团队里的其他求职者信任其思路, 他们才愿意配合领导一起来充实这个解决思路。这个思路不一定全部由自己提出, 可以综合众人心智。领导可以在引导和总结其他求职者思路的时候, 体现自己的领导能力和团队合作能力。

(2) 计时员。计时员的主要任务是进行时间管理, 在讨论过程中, 要严格按讨论好的时间规划来管理时间, 适当打断发言超时的同学。例如, “大家注意时间, 有点超时”、“我觉得××说得很有道理, 但是由于时间有限, 我们还是听听下一位同学的意见吧”、“×××同学, 你的发言时间完了, 请下一位发言”等。

(3) 组员。普通组员的主要任务就是进行项目的讨论, 将自己的观点准确无误地提交整个小组讨论, 对于组内其他求职者提出的观点也可以进行意见交换, 对于有异议的观点, 互相交换看法, 做到该说的就说, 不该说的千万不说。

(4) 记录总结员。记录总结员一般需要标记讲话内容的重点。一般而言, 记录总结员需要做到以下几点: 首先, 记录清晰, 重点标明, 能够快速而准确记下每个求职者的发言要点, 并结合团队整体解决思路, 把相关的发言重点, 用记号标明。其次配合领导, 推进讨论, 及时把要点清晰地指给或传给领导看。当团队成员讨论无目的或者偏离要点时, 记录总结员需要及时地将话题引入到正确轨道上。最后, 总结发言, 当大家都发言完毕时, 进行总结发言, 将整理出来的方案要点逐条讲出来, 在此过程中, 恰当点名赞扬一下某个同学的点子。

在确定好了自己扮演的角色之后, 接下来的就是技巧了, 在群面中一般还需要注意以下几个方面的内容:

首先, 对于一个话题, 小组成员应该有自己的观点和主见, 与小组其他人或其他小组人的意见可能相同或相似, 也可能存在意见不一致的情况, 所以当与别人意见一致时, 可以适当阐述自己的论据, 补充别人发言中存在的不足之处, 而不应该简单地附和说: “某某已经说过了, 我与他的看法基本一致, 我没什么好说的了。”这会给人一种没有主见、没有个性、缺乏独立精神的感觉, 甚至还会怀疑你其实根本就没有自己的观点, 有欺骗的可能。同时, 当别人发言时, 应该学会倾听, 目光注视对方, 不要有不自觉的小动作, 更不应该因为对对方观点

不以为然而显出轻视、不屑一顾的表情，这样会给人一种轻浮的感觉。对于别人的不同意见，也不要打断对方的发言或是抢问抢答，生怕别人不知道你反对，而是应在对方陈述完毕之后，很自然地发言，沉着应付，不要感情用事，保持清醒的头脑，思维敏捷，更利于分析对方的观点，阐明自己的见解。要以理服人，尊重对方的意见，不能压制对方的发言，不要全面否定别人的观点，应该以探讨、交流的方式在较缓和的气氛中，充分表达自己的观点和见解。

其次，在双方交谈的过程中，求职者也应该注意自己的态度和语气。自命清高、装腔作势、喋喋不休的人，不但不能引起面试官的好感，反而会给人留下傲慢、自私的印象，破坏交谈的气氛，很难达到彼此沟通的目的，从而影响到面试结果。

正确的做法主要包括以下 6 点。第一，充满自信，放下包袱，大胆开口，群面虽然是众多求职者之间的较量，但它本身并不可怕。每个人都是公平的，对于每个求职者而言，如果胆小怯场、沉默不语、不敢放声交谈，那就等于失去了被面试官考查的机会，很难被面试官认可。第二，讲话注意语速和音调，不要遇到激动的事情就抬高嗓门，遇到不确定的事情、心虚的时候就小声嘀咕，停顿时应该显得像是在思考的样子。第三，论证充分，辩驳有力，千万不能夸夸其谈、不着边际、胡言乱语。语不在多而在于精，而且言多必失，观点鲜明、论证严密，可起到一鸣惊人的作用。当表达与别人不同的意见和反驳别人先前的言论时，不要恶语相加，既要能够清楚表达自己的立场，也不要令别人难堪，等对方回答完毕后再回答，切忌打断他人说话。第四，尊重队友，友善待人，每一个求职者都希望抓住机会多发言，以便表现自己，而过分表现自己，对对方观点无端攻击、横加指责、恶语相向，往往会给面试官一种不重视合作、没有团队精神的感觉，而在团队中，有这种人的存在是要不得的。第五，不搞“一言堂”，不可滔滔不绝、垄断发言，也不能长期沉默、处处被动，每次发言都必须有条理、有根据。第六，准备手表和纸笔，记录时间和要点，随身携带一个手表和小笔记本，在别人滔滔不绝地讨论时，你可以做些记录，表明你在注意听，而不是“事不关己高高挂起”的心态。

常见的群面题形式多样，但万变不离其宗，以下 3 个题目就是一些大型的 IT 企业中出现过的群面真题：1) 如果唐僧去西天取经，可以带 8 个人去：李魁、孔子、瓦特、林黛玉、郑和、武则天、牛顿和李白，请你把这 8 个人按照你想带的意愿从强到弱排个序，并解释为什么这么排序。2) 做一个成功的领导者，可能取决于很多的因素，如善于鼓舞人、能充分发挥下属的优势、处事公正、能坚持原则又不失灵活性、办事能力强、幽默、独立有主见、言谈举止有风度、有亲和力、有威严感、善于沟通、熟悉业务知识、善于化解人际冲突、有明确的目标、能通观全局、有决断力，请你分别从上面所列的因素中选出一个你认为最重要和最不重要的因素。3) 各位乘坐的轮船触礁，15 分钟内要转移到荒岛，船上有指南针、剃须镜、饮用水、蚊帐、压缩饼干（一箱）、航海图（一套）、救生圈（一箱）、柴油（10 升）、小收音机（一台）、驱鲨剂（一箱）、20m² 雨布一块、二锅头（一箱）、15 尺细缆绳、巧克力（1kg）、钓鱼工具（一套）、火柴、香烟，哪些要优先带走。

只要掌握了以上提出的一些群面技巧，任何群面题目都能迎刃而解。

4.1.15 如何准备电话面试

由于求职者众多，而且很多求职者的个人简历中的“水分”也越来越大，如何在众多简历中挑选出具有真材实料的人才是每一个企业都面临的巨大问题，所以为了在面试前做进一步的筛选，用人单位往往以打电话的形式对求职者进行首轮面试，从而进行初步筛选。

电话面试的时间一般会持续 20~30 分钟，面试官通过短暂的电话交流用以核实求职者的背景和语言表达能力，电话面试不像面对面交流时那样直接，表现余地也相对较小，所以对于

求职者来说，如何在短暂的时间、非面对面的环境中脱颖而出是每一个电话面试求职者都值得深思的问题。

对于电话面试，求职者需要以饱满的热情、积极的心态坦然面对。首先，当求职者接到企业打来的电话时，可能因为正在上课或者正在运动，或者正在公交车上而无法正常工作交流，最好首先试探看看对方是否可以给一些准备时间稍后再进行电话面试，如“对不起，我正在有事，能不能换个时间给您打电话？”等，最忌讳说自己没有准备，即使是真的毫无准备，也不能如此回答，否则可能会引起面试官的反感，认为求职者不重视该企业给予的面试机会，导致最终失去机会。而一旦赢得时间，就需要赶紧准备，马上摊开求职资料写一份提纲，从容应答，最好预先准备好简历上的东西，确保能够清晰流畅地说出，还可以事先准备一下可能会遇到的问题，有备无患。然后找一个安静的环境，确保手机信号畅通，声音清晰，电量充足。坦然放松地与对方进行电话交谈时，应该将对方的单位名称、招聘岗位，以及你所感兴趣的职位等弄清楚。

其次，电话面试的内容一般是确认简历的真实性，看看是否有漏洞，是否有失实的地方。回答过程中的任何犹豫都有可能给对方造成说谎的印象，因此最好将简历放在手边，可以看着内容回答提问。对简历内容确认之后，面试官会针对应聘岗位问些专业技术方面的问题，如专业技能、对应聘职位的看法等，对于这些问题，不要慌张，抓住问题要点，尽可能如实回答。

再次，就是接听电话要冷静，注意语速，不要说话太急，不要夸夸其谈。在回答一些专业问题时，要尽量显示对那些专业术语非常熟悉，并能用简短的语言表达清楚、重点突出，不要回答得含糊不清。面试官可能会对求职者提出各种五花八门甚至让求职者感觉很“偏”的问题，以此来衡量你是否适合本公司，同时求职者也可以向面试官提出任何你想了解的问题，但待遇问题是一个“雷区”，最好不要提及，否则面试官会认为你比较功利。在面试过程中不要机械地背诵准备的材料。回答问题时语速不必太快，要简明扼要，发音吐字要清晰，表述要简洁、精辟，不拐弯抹角，不要使用宣誓、方言或行话。由于是电话面试，声音会受到多种因素的影响，如果问题确实没有听清楚，求职者要很有礼貌地请面试官重述一次，不要不懂装懂、似懂非懂、答非所问，如有必要，甚至还可以要求面试官改用其他方式重述他的问题。回答问题的时候还可以在旁边放杯水，口渴时候润润喉，如果能够准备计算器与一定的工具书在身边就更好不过了，不要吸烟、吃东西或嚼口香糖。使用某某先生或女士等称谓，时不时在对话中以此称呼对方，也可以数次提及公司的名字，不过不要去姓留名地称呼对方，除非对方提出你可以。保持微笑，微笑会改变说话者的声音，会给面试官留下积极乐观的印象。

最后，注意礼貌。当电话结束时，要记得感谢面试官，显示你的职业素养，可以这么说：“感谢您的来电，谢谢您对我的认可，希望能有机会与您面谈，您有任何问题可以随时给我打电话，打扰您了，谢谢。”如果对方直接约定面试，一定要拿笔记下时间、地点，重复一次，保证准时参加面试。

4.2 从容应对

在有限的时间，面试官们也很难提出既有新意，又有技术含量的问题，因为他们关注的问题往往就是那些常见的问题，确切地说，尽管面试官千千万万，面试题却是万变不离其宗：绕来绕去就是那么几道题，除了技术以外，无外乎就是关注求职者的性格、人品是否能胜任岗位，所以有针对性地在面试前进行充电完全能够应对绝大多数面试的需要。

4.2.1 如何进行自我介绍

自我介绍是面试中至关重要的一个步骤，很多面试官对求职者提出的第一个问题往往就是“请你先自我介绍一下”。有时候求职者会对此很困惑，个人情况在简历里面已经写得很清楚了，为什么几乎所有的面试官都要让求职者来做一个自我介绍，这不是多此一举吗？自我介绍看似简单，其实不然，面试官希望通过面试中的自我介绍环节来考查求职者以下几个方面内容：

(1) 考查求职者是否诚实。一般而言，如果简历的内容是真实可信的，口述自我介绍时就不会有明显的出入，但如果简历有假或者“水分”比较多，那么自我介绍阶段一般就会有破绽。此时，如果求职者反问面试官：“简历里面都写清楚了”，此时，面试官对求职者的印象分会很低。

(2) 考查求职者是否具有敏锐的逻辑思维能力、良好的语言表达能力、精练的总结概括能力。

(3) 考查求职者是否具有现场的感知能力与把控能力。

(4) 考查求职者的自我认知能力和价值取向。自我介绍本身就是求职者对自己各方面的一个归纳总结，同时会表达一定的个人价值取向。

(5) 考查求职者的理解能力以及时间掌控能力。有时面试官给出的问题是“请您用3~5分钟做一个自我介绍”，而求职者有时一介绍就滔滔不绝，刹不住了，往往超过10分钟，甚至20分钟，逼得面试官不得不多次提醒引导，最终当然会降低面试官的好感。

看起来，自我介绍是一个求职者被面试官考查的过程，完全处于被动状态，其实也不见得，求职者也可以化被动为主动。因为自我介绍也是一个求职者向面试官自我展示的平台，求职者可以通过自我介绍向面试官展示自己的能力和才华，向面试官推销自己，提升自己在面试官心目中的第一印象。

具体而言，面试官能够接受的自我介绍时间一般不会很长，太短不利于求职者介绍自己，太长会给人拖沓冗余的感觉，所以一般在3分钟左右。在这3分钟时间里面，求职者自我介绍的内容一般应该包括以下几个方面：

(1) 基本信息。通过个人基本信息的介绍，让面试官明白坐在他对面的人到底是谁。个人基本信息一般包括姓名、籍贯、年龄、教育背景以及与应聘职位密切相关的一些个人特长等。一般为了使得面试的氛围变得轻松，求职者可以采用一些生动、幽默、个性化的介绍方式。例如，我叫程浩，很好记的一个名字：加号、减号、除号、“程浩”，并且作了一个双手交叉在胸前的手势。诸如此类的方式不仅能够准确地介绍自己的基本信息，还能缓解面试初期的紧张气氛。

(2) 实践经验。实践经验记录了求职者的经验和经历。在此部分，求职者主要介绍与应聘职位密切相关的实践经历，包括校内外活动经历、相关的兼职和项目经验、社会实践等情况。明确这些实践发生的时间、地点、担任的职务、采用的技术、工作内容、工作量等信息，这样让面试官觉得真实、可信，没有弄虚作假的嫌疑。同时需要指出的是，当求职者的经历比较多时，很难做到面面俱到，那些与应聘职位无关的内容，即使你引以为荣也要忍痛舍弃。例如，应聘软件研发岗位，可以将支教、扶贫等与此无关的内容删除，尽量选择一些与软件开发相关的实践写进来。

(3) 成果展示。成果展示代表了求职者的能力和水平，在此部分，主要进行与求职者能力相关的个人业绩、获奖情况、校内外活动成果等展示，需要把各个阶段有代表性的事情描述

清楚（除非高中阶段有过人的成绩，一般从大学开始记录）。

在进行成果展示的时候，需要注意以下几个方面的内容：

- 1) 除了个人成绩以外，一般还应该包括团队成绩，如数学建模大赛、程序设计大赛等。计算机成果已经远非一个人能完成的，一般都是依靠一个团队完成的。
- 2) 内容有所侧重，要着重介绍那些能体现自己能力的重点，如华为杯软件设计大赛、腾讯创新设计大赛、中兴捧月程序设计大赛等与专业相关的竞赛应该仔细介绍。
- 3) 巧设伏笔，引导面试官向自己擅长的方面提问。例如，在介绍成果时，可以这样描述：“在开发过程中遇到了很多的问题，不过我还是成功地克服并达成了业务目标。”引导面试官提问“遇到了哪些问题”，然后你就可以进一步阐述细节内容，体现出自己处理问题的能力。
- （4）职业规划。职业规划代表着求职者的职业理想。在此部分求职者应该介绍自己对应聘职位、行业的个人看法和个人规划，包括求职者的职业生涯规划、未来的工作蓝图、对工作的兴趣与热情、对行业发展趋势的看法等内容。同时，求职者还要针对应聘职位合理编排每部分的内容。与应聘职位关系越密切的内容，介绍的次序越靠前，介绍得越详细。

在自我介绍时，还可以适当地介绍个人爱好等方面内容，如业余喜欢打篮球、爬山等。

需要特别强调的是，自我介绍时，对那些需要列举数据的地方要特别注意，不要与自己的个人简历表格上的内容有冲突。同时也不能在自我介绍的时候记流水账，要有亮点，给面试官留下鲜明的印象，更不能主动提及个人的缺点、弱点，缺点、弱点虽然可能会让面试官觉得自己很坦诚，但是一旦让面试官觉得这些缺点会使你无法胜任应聘职位的话，就得不偿失了。

以下是一个自我介绍的模板。

各位老师，大家上午好，我叫××，是×××大学×××学院的硕士研究生，本科是×××大学×××专业，今年××岁，我来自湖北仙桃，不是吃的仙桃，是地名。

今天我来应聘的岗位类型是软件类研发，之所以选择软件研发作为职业生涯的起点，是因为我对编程有着浓厚的兴趣。大学期间，除了学习书本上的专业基础知识以外，我也积极提高自己的专业技能与综合能力。利用业余时间，我参加过各类学科竞赛，曾经获得过数学建模二等奖、软件创新设计大赛二等奖、程序设计大赛三等奖的成绩，通过学科竞赛不仅扩展了我的专业知识面，更加使我意识到了理论知识只有应用在实践中才有意义。所以，在学校期间，我一方面认真上课，学习更加高深的计算机专业理论知识；另一方面参与到实验室的重要项目中，配合老师和同级其他同学一起编写代码、测试程序、编写相关文档。

软件开发是一个团体性行为，而且软件技术日新月异，每当在实际的开发过程中遇到困难时，我首先想到的是通过自己的独立思考来解决，我觉得很多时候只要自己多查阅一些相关文献资料或者网络资源，问题都能迎刃而解。自己无法解决时，我也会求助于老师与学长，学习他们的思维方式与解决问题的办法。通过多个项目的参与，我的编程能力、独立思考问题能力、团队合作能力都有了很大的提高。

学习需要劳逸结合，课余时间，我喜欢参加一些体育活动。在2010年11月我组织了学院的运动会，并取得了圆满成功。对于未来，我充满了信心，也感受到了一定的压力，万丈高楼平地起，我希望自己能够在自己喜爱的软件研发中找到一片属于自己的天空，在技术上有所提高，能够独当一面，成为一名对企业有用的人。

4.2.2 你对我们公司有什么了解

回答这类问题的时候千万要谨慎，不要泛泛而谈，如果求职者的答案是完全不了解，那么就没有必要继续面试了，当然录用的可能性也几乎就为零，没有哪一家企业喜欢对自己一无

所知的人成为自己企业的一员，求职者被录用的机会当然也就很渺茫了。一定要有备而来，事先做好“功课”，多了解一些与企业有关的信息，最好能够表达出对企业有关方面非常感兴趣，所以求职者必须能够谈论关于这个企业的产品、业界声望、服务、形象、目标、管理风格、历史和企业文化等问题，但是也不要表现出对这个企业的一切都了如指掌，最好的回答能够体现出自己对该公司做了一些研究，同时表达出希望能够了解关于公司更多的情况的愿望。

针对这样的问题，可以采用这样的态度来开始回答：“我觉得贵公司是最感兴趣的公司之一。在我求职的过程中，我也对比过很多其他同类型企业，我觉得贵公司在员工管理、人才培养、薪酬待遇等方面都非常好，出于这个理由，我还是更加倾向于贵公司。”

4.2.3 如何应对自己不会回答的问题

在面试的过程中，求职者对面试官提出的问题并不是每个问题都能回答上来，计算机技术博大精深，很少有人能对计算机技术的各个分支学科了如指掌，而且抛开技术层面的问题，在面试那种紧张的环境中，回答不上来的情况也容易出现。面试的过程是一个和面试官“斗智斗勇”的过程，遇到自己不会回答的问题的时候，错误的做法是保持沉默或者支支吾吾、不懂装懂，硬着头皮胡乱说一通，这样会使面试气氛很尴尬，很难再往下继续进行。

其实面试遇到不会的问题是一件很正常的事情，没有人是万事通，即使对自己的专业有相当的研究与认识，也可能在面试中遇到感觉没有任何印象、不知道如何回答的问题。在面试中遇到实在不懂或不会回答的问题，正确的办法是本着实事求是的原则，态度诚恳，告诉面试官不知道答案。例如，“对不起，不好意思，这个问题我回答不出来，我能向您请教吗？”

征求面试官的意见时可以说说自己的个人想法，如果面试官同意听了，就将自己的想法说出来，回答时要谦逊有礼，切不可说起没完。然后应该虚心地向面试官请教，表现出强烈的学习欲望。

所以，遇到自己不会的问题时，正确的做法是：“知之为知之，不知为不知”，不懂就是不懂，不会就是不会，一定要实事求是，坦然面对。最后也能给面试官留下诚实、坦率的好印象。

4.2.4 如何应对面试官的“激将法”语言

“激将法”是面试官用以淘汰求职者的一种惯用方法，它是指面试官采用怀疑、尖锐、咄咄逼人的交流方式来对求职者进行提问的方法。例如，“我觉得你比较缺乏工作经验”，“我们需要活泼开朗的人，你恐怕不合适”，“你的教育背景与我们的需求不太适合”，“你的成绩太差”，“你的英语没过六级”，“你的专业和我们不对口”，“为什么你还没找到工作”，“你竟然有好多门课不及格”等，很多求职者遇到这样的问题，会很快产生我是来面试而不是来受侮辱的想法，往往会被“激怒”，于是奋起反抗。千万要记住，面试的目的是要获得工作，而不是要与面试官争个高低，也许争辩取胜了，却失去了一份工作。所以对于此类问题求职者应该进行巧妙的回答，一方面化解不友好的气氛，另一方面得到面试官的认可。

具体而言，受到这种“激将”时，求职者首先应该保持清醒的头脑，企业让你来参加面试，说明你已经通过了他们第一轮的筛选，至少从简历上看，已经表明你符合求职岗位的需要，企业对你还是感兴趣的。其次，做到不卑不亢，不要被面试官的思路带走，要时刻保持自己的思路和步调。此时可以换一种方式，如介绍自己的经历、工作、优势，来表现自己的抗压能力。

针对面试官提出的非名校毕业的问题，比较巧妙的回答是：比尔盖茨也并非毕业于哈佛大学，但他一样成为了世界首富，成为举世瞩目的人物。针对缺乏工作经验的问题，可以回

答：每个人都是从没经验变为有经验的，如果有幸最终能够成为贵公司的一员，我将很快成为一个经验丰富的人。针对专业不对口的问题，可以回答：专业人才难得，复合型人才更难得，在某些方面，外行的灵感往往超过内行，他们一般没有思维定势，没有条条框框。面试官还可能提问：你的学历对我们来讲太高了。此时也可以很巧妙地回答：今天我带来的 3 张学历证书，您可以从中挑选一张您认为合适的，其他两张，您就不用管了。针对性格内向的问题，可以回答：内向的人往往具有专心致志、锲而不舍的品质，而且我善于倾听，我觉得应该把发言机会更多地留给别人。

面对面试官的“挑衅”行为，如果求职者回答的结结巴巴，或者无言以对，抑或怒形于色、据理力争，那就掉进了对方所设的陷阱，所以当求职者碰到此种情况时，最重要的一点就是保持头脑冷静，不要过分较真，以一颗平淡的心对待。

4.2.5 如何处理与面试官持不同观点的问题

在面试的过程中，求职者不可能所有的观点都与面试官一模一样，很有可能对某个问题的看法两个人相去甚远。当与面试官持不同观点时，有的求职者自作聪明，立马就反驳面试官。就算与面试官持不一样的观点，也应该委婉地表达自己的真实想法，因为我们不清楚面试官的度量，碰到心胸宽广的面试官还好，万一碰到了“小心眼”的面试官，他和你较真起来，吃亏的还是自己。

所以回答此类问题的最好方法往往是应该先赞同面试官的观点，给对方一个台阶下，然后再说明自己的观点，用“同时”、“而且”过渡，千万不要说“但是”，一旦说了“但是”，就容易把自己放到面试官的对立面去。

4.2.6 如果你在这次面试中没有被录用，你会怎么办

求职是一个双向选择的过程，有接受就有拒绝，有成功就有失败。所以，作为一名求职者，一定要清楚这一点，有时候，即使自己的实力达到了企业的标准，也不能保证万无一失，最终也有可能被企业拒绝，成为计算就业率时候的分母。

当面试官对求职者提出此类问题的时候，并不表示求职者就没有希望了，就一定不会被录用。一般而言，没有哪个面试官真的是因为要拒绝求职者才提出此类问题。如果真要拒绝求职者，也不用多此一举，提出此类问题了。提出这类问题的目的，主要是想考查求职者在遇到挫折时的一种应对措施，以此评价求职者的处事能力，毕竟未来的工作往往会有困难、有挫折。

作为面试官，一般希望求职者在遇到失败的时候，能够具备以下优良素质：

(1) 敢于面对。面对失败不气馁，从心理意志和精神上体现出对这次失败的抵抗力。

(2) 自信。相信自己经历了这次之后经过努力一定能行，能够超越自我。

(3) 善于反思。对于失败的教训能认真客观地总结，能够从自身的角度找差距，而不是怨天尤人。不要抱怨面试官不是“伯乐”，首先要看自己是不是“千里马”。能够正确对待自己，实事求是地评价自己，辩证地看待自己的长短得失，做一个明白人。

(4) 能够走出阴影。每一次失败都会给自己的心灵上抹上一层阴影，能克服这一次失败带给自己的心理压力，时刻牢记自己的弱点，防患于未然，加强学习，提高自身素质。

(5) 再接再厉，继续努力。能够在以后的学习工作中继续努力，争取取得下一次的 success。

4.2.7 如果你被我们录取了，接下来你将如何开展工作

面试官提出该类问题一方面是为了考查求职者的应变能力，另一方面也是为了考查求职

者是否有一个良好的规划，因为好的规划是成功的开始。

所以，在回答此类问题时，应该着重突出以下几个方面的内容：

(1) 个人适应能力强。强调自己能够尽快熟悉工作环境，融入工作集体，了解本单位的工作职能、组织架构以及自己的工作职责。

(2) 谦虚谨慎、低调做人，不自高自大。强调进入一个新单位，面临一个陌生的环境，自己作为一名新人，会有很多问题需要学习，自己一定会积极学习、虚心求教。向领导、向老同志、向同事学习，与他们多交流、多沟通，加深了解，增进感情；不争名夺利，不斤斤计较；识大体、顾大局，争取能够早日胜任新的工作。

(3) 服从安排。强调自己会遵守纪律，服从组织安排，在争取自己权益的同时，也顾全企业大局，脚踏实地地从事本职工作。

(4) 努力工作。强调自己尽快地进入工作角色，学习岗位相关知识，努力工作，以企业利益最大化为奋斗目标，不断开拓创新，为企业的美好明天贡献自己的一份力量。

如果求职者对于其应聘的职位没有足够的了解和对行业足够的洞察，不要直接说出自己开展工作的具体办法，可以尝试采用迂回战术来回答。比如，“首先会听取上级的指示和要求，然后就相关情况进行了解和熟悉，并在此基础上，制订一份近期的工作计划并报上级审批，最后根据审批后的工作计划开展本职工作”。

4.2.8 你怎么理解你应聘的职位

面试官提出这类问题，是想考查求职者对所应聘岗位的熟悉程度，以及对未来工作的一种认知程度。虽然求职者的回答不能完全反映其是否能够胜任招聘的岗位，但也能够基本反映出求职者对本行业、本公司、本岗位的了解程度。

针对这类问题，最好的回答要点是把岗位职责和任务及工作态度阐述清楚，而且回答应该简洁、明了，应该基于工作要求。在回答前，争取做到对这个职位的方方面面都有比较全面的认识。如果能够预先多掌握一些有关企业的资料，面试时描述清楚，一定会令面试官刮目相看，并且会认为你加入该企业的诚意无可置疑。如果能够做到对所应聘的职位的性质、工作内容、所需专业知识了如指掌的话，面试官将会更相信你比较适合所应聘的职位。但是如果确实不清楚或者有些方面的内容不确定，可以去询问面试官，他可能会帮助你回答这个问题。

例如，可以回答：“我应聘的岗位是云计算研发工程师，云计算是未来IT业发展的一个方向，我觉得未来云计算将会改变整个人类的生活状态，每个人都能从云计算中得到方便与实惠。××公司作为中国民营企业的杰出代表，能够进入世界500强，显示出了其强大的管理能力与研发能力。我平时也非常关注云计算相关的技术，参与过多个实际的云计算相关项目，对Hadoop开源框架也具备一定的了解，现在××公司将云计算作为未来的一个增长级，大力发展云计算，我感觉对我个人来说，将是一个巨大的机会，所以我非常希望能够加入××公司这个大家庭，施展自己的才华。”

4.2.9 你有哪些缺点

求职者最害怕面试官询问有关自己最大的缺点的问题，可是面试官却总是会对求职者提出此类问题，似乎非要让求职者在自己面前出丑。而作为求职者而言，最好的办法就是从容面对。

面试官通常不希望听到直接回答的缺点有很多，如求职者说自己小心眼、爱忌妒人、非常懒、脾气大、工作效率低、性格急躁、不注重家庭等，企业肯定不会录用你，同时也要避免

说出那些与求职岗位相冲突的缺点。例如，编程需要心细，可是却说自己的缺点是粗心，那么这份工作就不适合你了。

自作聪明的人的回答是：“我的缺点是没有任何缺点”、“我的缺点是过于追求完美”、“我最大的缺点就是优点太多了”等，往往会招至面试官的反感。

还有一种回答：“我个人不太注意家庭生活，是一个名副其实的工作狂，虽然我也知道疯狂工作会影响身体，但是为了工作能够完成，我也没有办法”，表面上看，这种人会非常讨面试官开心，但其实不然。凡是做出这种类似回答的求职者，在面试官看来，往往是那些没什么想法或者说分析思考能力较弱的人，再或者是一些有一定分析思考能力，但弄巧成拙的人。

其实，回答这类问题并不困难，每个人都会有缺点，只要你的缺点不与求职岗位相冲突，不是正常人看来的劣行品质，也不会影响到面试官对你人品的怀疑或是对你工作能力的怀疑。而且缺点不是绝对的，缺点是相对的，不同的职业对缺点的定义也不一样，所以回答缺点时，不要有任何心理负担。在回答该类问题时，常常需要注意以下事项：

(1) 不应该说没缺点。

(2) 不应该把那些明显的优点说成缺点。

(3) 不应该说出严重影响所应聘工作的缺点。例如，缺乏团队精神、承受压力的能力不强、缺乏领导能力、表达能力不强、过分追求完美等。

(4) 不应该说出令人不放心、不舒服的缺点。

(5) 可以说出一些对于所应聘工作“无关紧要”的缺点，甚至是一些表面上看是缺点，从工作的角度看却是优点的缺点。例如，① 我有时候做事情比较急于求成，一旦接受一个任务，总想最高效地完成，但是欲速则不达，有时候在追求高效的时候，却忽视了精确性；② 我做事情比较需要外界压力推动，压力越大，效率越高，但若是在压力小竞争强度小的环境下，有可能反而变得有些松散；③ 我缺乏工作经验，导致我做事情的时候不够自信。④ 为人处事，特别是处理上下级、同事间关系的经验还有待进一步提高。诸如此类回答，一般是面试官满意的。

有时候问完了求职者的缺点之后，面试官还会在此基础上引申一下，提出一个问题：“你是如何处理别人的批评的”。其实每个人在工作中，都不可能一帆风顺，都会经历到各种各样的挫折甚至是苦难，当遇到别人批评自己时，不管别人是出于什么目的，我们都应该虚心接受，然后及时修正与处理，不应该因为别人批评自己就怀恨在心甚至大动干戈。例如，可以回答：“对我提出批评的人，我并不怪他们，我觉得他们都是为了让变得更好，都是帮助我的人，我会有则改之，无则加勉”或者回答“人非圣贤，孰能无过，过而能改，善莫大焉，人在职场，面对领导、同事或者朋友的批评在所难免，只有虚心听取他人意见，才能认识到自己的不足，进而提高自身水平。”

4.2.10 你有哪些优点

相比回答最大的缺点，这个问题其实更有挑战性，回答得太直接，恨不得把自己从小到大所获得的奖项统统报出来，反而可能会给面试官一种夸夸其谈的感觉，而回答得太委婉，又突出不了优点，无法增加面试官对自己的印象分。

其实，面试官提出这类问题，主要是关注求职者的两点内容：第一，求职者的诚信；第二，求职者的素养，包括技术能力、为人处事能力、沟通能力等。

回答该问题的时候，应当首先强调自己已具有的技能。面试官是否雇用你很大程度上取决于你的这些技能。可以参考以下两种回答：

(1) 一旦有了目标,我就会朝着目标不断努力,而一旦我下定决心做某事,我就要把它做好,绝不拖拖拉拉、半途而废。例如,我的志愿是成为一个出色的系统分析师,我喜欢学习新的技术,关注最新的IT业发展,为了实现这个目标,我目前正在修读有关课程,并且参与到了一个实际的项目开发中。

(2) 我做事有计划,有安排,我觉得如果不好好规划,事情很难取得成功。所以每天早上起床,我的第一件事就是制订一个比较详细的计划,把当天要做的事情分为两类:必须要完成的与最好能完成的。在职业发展方面,从大三开始我就决定毕业后要从事互联网企业研发工作,所以平时尽量多参与到一些与互联网应用相关的项目中去,尽可能地学习到互联网软件开发技术。

例如,“我的学习能力”、“适应能力很强”、“人际关系很好”等都是可提出的优点,如果能够提供与工作相关的证据,将能起到锦上添花的效果,受到用人单位的青睐。

回答完自己的优点后,最好能够继续向面试官表达一个意思:虽然自己优点很多,而且也取得了一些成绩,但是这些都是历史,现在最希望的是在新的工作岗位上能够继续不断地完善自己。

当然,很多时候,有些个性对于某一个岗位可能是优点,对其他岗位可能是缺点。例如,对于研究性岗位,就需要那种偏内敛、坐得住的人;而对于销售岗位,则更青睐性格外向的人。

4.2.11 你没有工作经验,如何能够胜任这个岗位

针对工作经验的问题,面试官会提问:你是一名应届生,缺乏工作经验,如何能够胜任你所应聘的岗位?

其实对于应届毕业生,面试官通常知道他们缺乏工作经验,却偏要就此问题发问,总是搞得求职者“丈二和尚摸不着头脑”,不知道“葫芦里卖的什么药”。其实一般情况下,如果面试官提出此类问题,说明该企业并不真正在乎“经验”,如果他们希望招募有经验的员工,就不会让你这样一个没有任何工作经验的人来参加面试和笔试了,面试官主要是想考验一下求职者的反应能力。所以针对此类问题关键看求职者如何巧妙回答了,求职者回答什么并不重要,关键是看他们被“刁难”后的态度。

对这个问题的回答最好要体现出求职者的诚恳、机智、果敢及敬业。例如,可以回答“我虽然是一名应届毕业生,确实缺乏工作经验,但是在学校读书的时候,我一直利用各种机会参与到实际的项目开发之中。同时,我发现实际工作需求的知识远比书本知识丰富、复杂。但我有较强的责任心、适应能力和学习能力,而且我相信勤奋是任何困难的杀手,所以在实际的项目中我都能圆满地完成各项指定的任务,从中获取的经验也令我受益匪浅。请贵公司放心,我在校期间参与的项目获取的经验使我一定能胜任这个职位。”

4.2.12 你的好朋友是如何评价你的

在面试的过程中,面试官会经常提出此类问题,主要是为考查求职者的性格,想从侧面了解一下求职者与他人相处的能力。这种问题看起来似乎与求职者的工作没有任何关系,但实际上体现了用人单位不仅注重求职者的专业技能,而且注重他们的人品。

从另一个方面讲,回答此类问题也是一个表现自己的最好机会,所以一定要好好地把握这样一个机会。如果回答的时候不知道从何说起,或者需要思考一会儿才能回答,或者不经思考随便回答,都不会引起面试官的好感。

因为朋友一般不可能评论自己的工作技能，所以回答的时候应该重点谈论自己身上被朋友认可的一些品质和个性特征，最好与工作也相关。例如，友好、外向的性格，幽默感；可靠、忠诚、能保守秘密、诚信；适应能力强、有责任心、做事有始有终；有毅力、有抱负、有决心；能锲而不舍，奋发向上。这些评价都能够为自己加分。

以下几种回答方式都是非常能够吸引面试官的。

(1) 我的朋友都觉得我是一个心怀远大志向的人。例如，在大学期间，为了减轻家庭负担，我选择了课余兼职，这样我就可以一边学习一边工作。

(2) 朋友们都说我是一个随和、值得信赖的人，我认为朋友对我的评价还是比较客观的。这与我的交友原则是分不开的。首先，我觉得一个人应该诚信，孔子说“人而无信，不知其可也”，只有真诚待人，才能取得别人的认可。其次，我觉得做人要低调，谦虚谨慎，对朋友要平易近人，不能恃才高傲，要有亲和力。最后，我觉得应该对人主动热情，对自己可以帮到朋友忙的地方要竭尽全力去完成好，人际交往是双向的，只有双方共同付出，友谊之花才能天长地久。

(3) 我的朋友都说我是一个可以信赖的人。我一旦答应别人的事情，就一定会做到，哪怕是再苦再难。如果我做不到，我也不会轻易许诺。

(4) 朋友们都说我是一个比较随和的人，与不同的人都可以友好相处。在与人相处时，我总是能站在别人的角度去考虑问题，这样也能更好地为他人着想。

当然，人都是两面的，有优点，也有缺点，千万不要将一些可能影响到岗位录取的缺点说出来。

4.2.13 你与上司意见不一致时，该怎么办

当面试官提出此类问题时，一般出于以下两个原因考虑：首先，工作中常会出现与上司意见不一致的情况，通过提问，提前了解求职者的处理方式；其次，考验求职者的沟通能力以及对自己的角色定位。

在回答此种问题时，如果回答地过于有个性，可能会让面试官觉得不够职业、成熟，但如果回答得太八面玲珑，也有可能让面试官觉得太虚情假意、见风使舵、不够稳重。因此，在面试过程中，回答此类问题的原则是诚恳、谦虚、稳重。

具体而言，应该在回答的时候表达出以下 3 个方面的内容：首先，自查，无论是与上司还是同事，抑或是与下属意见不一致时，既不能对上趋炎附势、迎合谄媚，也不能对下盛气凌人、以权压人，要学会先从自己身上找原因，而不是无端地指责别人或是列举种种客观因素为自己开脱责任，尤其是当与上司意见不一致时，更不能贸然地去质疑对方，而是应该静下心来，重新梳理自己的思路，与对方的意见做仔细比较。然后，用心沟通，只有稳定情绪、心平气和的沟通才是沟通的最佳状态，“气急之下无好话”，气急也不利于观点表达，反而会让有理变无理。“有理不在声高”，即使自己的观点真的是正确的，也不能一副胜券在握、目中无人的傲慢态度，摆事实、讲道理，以谦逊的态度阐述自己的观点，并举例说明情况。最后，顾全上司的面子，给上司一个台阶下，顾及他人颜面也是一种美德，尤其是同事之间，特别还是上司下属之间，尽量不要在众多同事面前与上司产生激烈碰撞，挑战上司的威信、当面指责上司的错误都不利于团队合作，还会影响到其他同事，对以后的工作开展都会有不利影响，所以最好选择单独沟通或者是先邮件沟通。

所以，作为下属，如果与上司意见不一致，原则上应该尊重和服从上司的领导与安排，但不要使用“你是上司你说了算”这种表达方式，可以在私底下寻找合适的机会，以请教的口

委婉地向上司表达自己的真实想法，看看上司是否能改变想法，但如果上司没有采纳自己的建议，自己也同样会按上司的要求认真地去完成这项工作。具体可以参考以下几种回答方式：

(1) 作为企业的一员，在工作中，与上司在某些问题上产生分歧在所难免，但是大家最终的目标都是希望能够更出色地完成任务，使得公司的利益最大化。当与上司发生意见不一致时，可能是思考问题的角度和沟通方式上出了问题，所以我不会急于去与上司辩论，而会首先站在上司的角度去考虑问题，毕竟上司比我有更丰富的实战经验，他是站在大局、宏观的角度考虑问题，而我的视野可能更狭窄一些，考虑得不够全面。

(2) 如果遇到这种情况，我觉得有效沟通是解决问题的最佳方法。首先，我会向上司表明自己希望沟通的愿望和诚意，同时在沟通的过程中，换位思考，站在上司的角度去考虑问题，然后再阐释自己的理由。在与上司的沟通中，我会保持谦和的语气与实事求是的态度，即使自己可能更在理一些，也不会“得理不饶人”，尽量照顾上司的面子与企业的形象。

4.2.14 你能说说你的家庭吗

一般而言，企业不需要知道求职者的家庭具体情况，之所以面试时询问求职者的家庭问题，并非要窥探求职者的个人隐私，而是基于以下几个方面的考虑：

(1) 此类问题可以理解为一种亲切的问候，通过这种亲切的问候，可以拉近面试官与求职者的心理距离，缓解求职者的紧张情绪，方便求职者正常发挥。

(2) 通过了解求职者的成长环境，包括父母职业、家庭成员构成等情况，做基本的家庭教育情况和家庭经济状况推测，来初步判断求职者是个什么样的人。

(3) 对求职者回答的内容进行更深入的追问，了解父母的价值观以及求职者个人的价值观，以及求职者对父母，父母对求职者的评价、态度、认同程度。同时，父母的人生观、价值观会对子女有深刻影响。

(4) 了解求职者与父母、亲人的相处情况，和谐、积极的家庭氛围对子女性格完善更有益处，以后工作中也更善于与他人合作、相处。

(5) 企业对求职者婚姻状况、子女状况有时会有倾向性的。例如，有些企业就希望招已婚已育的女性。

企业希望听到的重点也在于家庭对求职者的积极影响，所以在回答此类问题时，一般需要考虑以下5点内容：第一，简单罗列家庭人口、父母职业等情况。第二，强调温馨和睦的家庭氛围，在真实的基础上尽量表达积极、正向的内容，不要过多提及父母感情不好等对自己不利的情况，即使单亲家庭，也可以从与母（父）亲相互鼓励、体谅、自强不息等方面着手，因为企业相信，和睦的家庭关系对一个人的成长有潜移默化的影响。第三，强调父母对自己教育的重视。第四，强调家庭成员的良好状况及其对自己工作的支持，以及自己对家庭的责任感。第五，强调自己非常热爱父母、热爱自己的家庭。

需要特别注意的是，在回答此类问题时，言语中应该尽量少用评价性、赞扬性的词汇，要用中性词和情感描述性词汇，因为你谈论的对象是你的家人，而不是下属，不能以过于理性评价或判断的姿态出现，否则可能被误认为没有谦卑和敬畏之心，或者认为过于虚荣，回答虚假。

对于家庭条件相对比较好的求职者而言，最好强调家庭条件对自己的个性形成起到积极作用；对于家庭条件相对不好的求职者而言，最好能够强调家庭条件促成了自己优秀品质的形成，表达出“穷人的孩子早当家”、“将相本无种，白屋出公卿”意思。有以下一些比较好的回答方式可供参考：

(1) 我很爱我的家庭，我的家庭一向很和睦，虽然我的父亲和母亲都是普通人，但是从

小我就看到我父亲起早贪黑，每天工作特别勤劳，他的行动无形中培养了我认真负责的态度和勤劳的精神。我母亲为人善良，对人热情，特别乐于助人，所以在单位人缘很好，她的一言一行也一直在教导我做人的道理。

(2) 我的家境不是很好，父母都是普通农民，虽然他们给予我的物质生活不是很好，但是我觉得已经足够了，他们朴实无华、勤勤恳恳，用他们辛勤的劳动来供我读书，非常不容易，所以我也非常能够体会为人父母的不容易，珍惜来之不易的读书机会。从小我也养成了吃苦耐劳的习惯，独立意识与适应环境的能力比较强，无论遇到什么困难，我都能非常从容地去面对。“海阔凭鱼跃，天高任鸟飞”，父母淳朴的性格永远指引着我不断前行，使我受用终身。

(3) 我的家庭背景还可以，父亲是政府公务员，母亲是大学教师，所以从小家教就比较严格，我也不会更多计较薪金上的问题，做事情更加看重的是能否体现个人价值、发挥个人能力，更加关注自己的未来发展方向，这也是选择贵公司的主要原因，毕竟这里代表着这个行业的发展方向，我希望能够在这样一个有发展前景的行业、有发展潜力的企业工作。

4.2.15 你认为自己最适合做什么

面试官提出此类问题，主要是想考查求职者的主观能动性。

针对此类问题，错误的回答是“只要公司需要，我什么都能干”、“你们需要什么我就能干什么”、“公司安排我做什么我就做什么”等，这类回答一方面会给人一种太随意、没有主见的感觉；另一方面，如果求职者什么都可以做，那还要其他人干什么呢？

在面试官心中，一个有活力的员工，必定是有追求、有理想、有抱负、有能力、并能脚踏实地工作的人。所以，此时既要向面试官“毛遂自荐”，也不要“谦虚过头”，把握有度，否则会面试官一种对自己的职业目标定位不明确的错误印象。

所以，自己适合做什么，就老老实实在地告诉面试官，实事求是，因为面试官也希望得到一个明确的答案，而且明确的答案也可以给人一种有思想、有主见、有活力的印象。如果真的没有硬性要求，也可以回答“服从需要”。但如果有需求，却为了讨好面试官，言不由衷，说出“服从需要”的空话来，最后有可能会被分配到了一个完全不适合自己的工作岗位上，影响自己日后的前程。

4.2.16 你如何看待公司的加班现象

由于软件行业发展迅速，而且技术更新快，为了尽快推出一款新产品抢占市场、占领行业的制高点，很多 IT 企业都会给程序员分配过多地任务，导致的直接后果就是程序员不得不通过加班来完成这些“超额”任务，加班、工作累已经慢慢成为程序员的标签。

所以，在 IT 企业的面试环节中，面试官一般会针对加班问题对求职者进行提问，当然面试官提出此类问题，也并不说明一定需要加班，他只是想测试求职者是否愿意为公司奉献。同时，IT 企业加班已经成为一个普遍现象，很少有不加班又高薪酬的企业，所以遇到此类情况，求职者首先需要明白的是无理加班不一定是最好的，一般回答“在自己的职责范围内进行的工作，不能称之为加班”、“如果是工作需要我会义不容辞地加班。我身体很好，没有任何负担，可以全身心地投入工作。但同时，我也会提高工作效率，减少不必要的加班”或者“我对加班是这样看的，既然来工作，就必须要有责任心，所以如果是因为工作需要而加班，当然没问题。但是也应该注意提高工作效率，如果是因为工作拖沓而加班，那是不可取的。例如，如果别人 8 个小时做完的工作我 4 个小时就做完了，那还为什么要加班呢？”比较能被面试官认可。

还有一种回答方式，可能有说谎话、谄媚、讨好面试官之嫌，但却更加能够获得面试官的青睐，就是强调加班是因为企业发展好，对于个人成长非常有用，对加班持完全肯定的态度。例如，可以回答如下：1) 如果工作确实非常紧急，真要在规定的日期内完成不可，那我就义无反顾加班。我个人觉得，如果项目没有完成，就算是节假日或是晚上，我也不能休息得踏实。而且，如果连节假日或是晚上都需要加班的话，说明项目确实是非常紧急，我所从事的工作一定是相当充实的，只要是有意义的工作，与其在家休假，还不如抱着一颗学习的心态高高兴兴地加班。2) 对于加班，我觉得累并快乐着，为什么要加班？说明企业效益好，蒸蒸日上，正处在高速发展并迅速提升的阶段，需要人手，在这样一个有着美好前景的企业里面工作，我感到前途一片光明。而且我很年轻，精力充沛，正是学知识、学能力、积累工作阅历的时候，虽然加班有可能让我感到疲惫，但我明白一个道理：没有太上老君的八卦炉就无法练就孙悟空的火眼金睛，它们就是熔炼我的火炉，只有韬光养晦，才能百炼成钢，干出一番事业。这种回答方式不只局限于是否能加班的问题，而且更进一步从加班谈到工作的充实感，独具匠心。

当然，可以就是可以，不行就是不行，如果确实不能接受加班，也不要为了博得面试官的好感而勉强自己，诚实回答，将自己的真实想法得体地表述出来不失为一种好的方法。

4.2.17 你的业余爱好是什么

面试官对求职者提问有关业余爱好的问题，首先是借此活跃一下严肃的面试气氛，其次是想看看对方的心理状态，一般而言，有业余爱好的人，即使遇到压力也可以通过个人爱好予以缓解，有利于快速适应陌生的环境以及紧张的工作。

虽然面试官问的是业余爱好，但是求职者在回答时也不能说得太随便，尤其是说一些庸俗的、令人感觉不好的爱好，如吃喝、泡酒吧等，都不合适，有些甚至会引起面试官的反感。但也不能说自己没有业余爱好，没有爱好的人，往往不会生活，会给人一种刻板、无趣的感觉，没有谁愿意与这种人一起工作。

同时需要注意的是，求职者在描述自己的业余爱好时，也不要“高大全”，泛泛地罗列一些不属于自己爱好的内容，一定要注意少而精。在面试过程中，求职者在回答自己的特长以及爱好时，有时可能会出现问题。例如，当求职者提到自己的爱好时，面试官也许比较感兴趣，也许会有异议，进而要求求职者展开描述，如果回答得不合适或不完善，很有可能会影响到面试的最终结果。

由于IT企业都需要团队合作，所以也不能说一些让面试官联想到性格孤僻的爱好。例如，说自己的爱好是读书、听音乐和上网。最好说一些实际的，能在一定程度上反映求职者的性格、观念、心态、能体现团队合作精神的的活动。例如，游泳、攀岩、打篮球或者户外运动等，这些都能体现出一个人的精神风貌：勇敢、开放、乐观、大气、创新等。

例如，可以回答如下：

我的业余爱好比较广泛，主要有以下几个方面的内容。

(1) 读书。我比较喜欢看书，尤其是历史书籍，喜欢从书中汲取营养，不断完善自己的人格，全面提升自己的综合素质。

(2) 公益活动。我喜欢参加一些公益活动，我觉得参加公益活动是一件非常有意义的事情，所以我会不定期地去献血。

(3) 演讲。我喜欢参加各类演讲比赛，也取得了一定的成绩，但我更注重的是演讲的过程，我觉得通过演讲，既锻炼了自己的口才，又学到了许多新的知识，同时还认识了很多好朋友。

(4) 体育运动。对于搞 IT 的人而言,每天对着计算机,对身体各个方面都有比较大的损伤,所以我觉得锻炼身体是对自己、对工作的最大负责。要想做好工作,健康的身体是一个必不可少的条件,所以我会每天都进行跑步、打篮球等各项体育运动。

(5) 唱歌。我喜欢唱歌,无论是独自一人清唱,还是与同学、朋友一超唱,我都非常喜欢。因为每当唱歌的时候,我不仅能领略到歌词的美妙,还能从繁忙的学习中解脱出来,心情愉悦。

(6) 爬山、旅游。我喜欢爬山,也喜欢旅游,爬山让我领略到了美好的自然风光,旅游让我体会到了异域情怀,通过爬山、旅游,使我深刻明白了一个道理:“读万卷书,不如行万里路”,用心去体会整个大自然、整个世界,生活确实无限美好。

4.2.18 你和别人发生过争执吗? 你怎样解决

面试官提出此类问题,真的是“用心良苦”,在这样一个“陷阱”中,求职者千万不要评价任何人的过错。对于一个团队而言,矛盾是一个团队必然存在的问题,再优秀的团队也不可能没有分歧或是矛盾,成功解决矛盾也是一个优秀团体中每个成员必备的一种能力。

在现实生活中,当与别人发生争执时,最明智的办法一般都是自己首先退让,缓和现场气氛,防止争执扩大化,对双方引起更大的伤害,等到合适时机,双方静下心来,心平气和地交换各自的想法,站在对方的立场考虑问题,最终互相让步,得到一种双方都能够接受的方案。

所以,对于该类问题,可以参考以下几种回答方法。

(1) 当我与别人发生争执时,首先我会调整自己的心态,坦然地去面对这个问题,客观地去看待问题,而不是选择逃避。如果我的立场是对的,我会委婉地拒绝争执而走开;如果对方的立场是对的,我也会主动认错,向对方道谢,并从中吸取教训。

(2) 当我与别人发生争执时,我一般会找对方坐下来面对面地讨论,询问他的想法,之后,我再讲出自己的观点。找出最重要的部分和需要妥协的部分,然后将观点中相同的部分加以明确,就需要妥协的部分交换意见。最终双方都会感到有所收获,讨论出满意的解决方案。

(3) 我觉得谁都有过与人争执的时候,当我与别人发生争执的时候,我会仔细分析问题出在哪里,如果是我的问题,我会主动承认错误,向对方道歉,并且接纳对方的合理观点;如果是对方的问题,我也不会抓住不放,得理不饶人,我会耐心地解释给对方听,而不是面红耳赤地和他争吵。

其实在工作中,一般都需要整个团队的协作才能更快、更好地完成任务,团队中的每个成员都是一个独立的个体,都有自己独特的审美观、价值观,加上每个人性格、阅历、生活环境、思考问题的方式等都不尽相同,所以在合作中,对于某个事物的看法有分歧在所难免,这是一个非常正常的情况。用争执来形容,似乎不太恰当,但不管怎样,大家都是在奉献自己的智慧,都是为了将工作做好这样一个共同目标在努力,是一个对事不对人的行为,因为大家还是在互相尊重,而且所谓争执仅仅是一个不同的看法而已,并不是很深的个人矛盾或者上升到敌对关系,所以也就不存在所谓的如何解决,如果非要说解决的话,那么加强沟通和配合上的默契才是最好的解决之道。

4.2.19 你如何面对压力

无论什么行业,都会有压力,而 IT 企业中的压力尤为突出,所以 IT 企业在进行面试的时候会对求职者进行该方面的提问,以确保求职者能够化解压力,有着健全的性格特征。

针对面试官的这种提问,求职者自身应该对工作中的压力有一个清醒的认识。首先自

身要有充分的思想准备,不管是一份什么工作,都要在入职前对它有一个详细的了解,包括工作的内容、强度,所要求的专业技能等。对工作有了了解后,才可以更好地准备,并以饱满的热情和精力来适应工作。即使压力非常大,如果事先已经有了心理准备,也可以从容应对。

其次,要学会自我调节。心态决定成败,当工作压力大时,应该学会调节。例如,喝一杯咖啡、听一首简单轻快的歌曲、到窗户边几分钟的眺望、写博客、给家人朋友打电话等。如果条件允许,还可以进行适度的体育锻炼,如跑步等。

最后就是给予自己乐观积极的心理暗示。“压力越大,动力越大”,所以在高压下工作,可以帮助自己在最短的时间有最大的提高,进而使自己的承受能力变强。

有时通过找有经验的人给出一些切实可行的解答和措施也可以缓解自身的压力。所以面对压力,当有了清醒的认识后,不应该惧怕,而是应该坦然面对,将自己积极的一面传达给面试官,从而得到面试官的认可。

4.2.20 你为什么离开了原来的单位

人都有一种固有的思维就是“这山望着那山高”,一般都认为最好的企业是下一个企业,但面试官提出此类问题,为了确认求职者不会再以相同的原因辞职。此类问题,如果回答的不合适,就会导致面试官觉得离开原单位完全是因为求职者个人原因造成的,而非因为工作上的原因,即使今天录用了求职者,求职者以后也可能会因为同样或类似的原因离开。

在回答该类问题的时候,避免把“离职原因”说得太详细、太具体,切忌掺杂主观的负面感受,切忌抱怨、诋毁以前的单位。例如,待遇低,与领导不和,与同事不和,不服从分配,工作太累、压力太大等,也不要涉及自己负面的人格特征,如懒惰、缺乏责任感、不诚实等。求职者应该更多地提及个人发展需要,而不是归咎于其他人,同时对于离开以前的单位,也应该表示很遗憾,情感上非常不愿意,以此让用人单位相信你具备出色的工作能力与良好的人际关系,能够胜任新单位的工作。

需要注意的是,除非是薪资太低,或者是第一份的工作,否则不要用薪资作为理由。“求发展”也被面试官听得太多,离职理由要根据每个人的真实离职理由来设计,但是在回答时一定要表现得真诚。例如,可以回答:“其实,在原来的单位工作了5年,与领导、同事相处非常融洽,彼此之间已经建立了深厚的感情,而且通过我的努力,也取得了他们的信任,他们也都非常不愿意我离开,我也很舍不得他们,作出决定离开那里,从情感上来说,我是非常痛苦的。但是,长久以来,我一直希望自己能够在云计算领域有所发展,由于一些客观原因,我未能在那里实现这个愿望,所以我还是做出了一个艰难的决定,最终选择了离开。”实在想不出来的时候,家在外地可以说是因为家中有事,需请假几个月,企业又不可能准假,所以辞职,这种答案也能被面试官勉强接受。

有时候,面试官可能还会在此问题的基础上引申一下,问你对跳槽有什么看法,此时也要小心,不要“中招”了。对于跳槽这个问题,最好能够表达两个意思:第一,正常的跳槽一般能促进人才合理流动,也是一种正常的行为,应该予以支持;第二,频繁的跳槽对单位和个人都不利,不提倡,应予以坚决反对。

4.2.21 你为什么更倾向于我们公司

这个问题也可以变化为“为什么你想到这里来工作”或“为什么选择我们公司”,面试官试图从中了解到求职者的求职动机、愿望以及对此项工作的个人态度。

这类问题一般在面试的后期被面试官提出。面试官提出这种问题，表明面试官已经对你有了一定的兴趣，同时希望具体看看你是否对他们公司也有着同样的兴趣。对于这个问题的回答，要格外小心，态度诚恳、谦和，无论是大企业还是小企业，都有其优势与劣势，但是求职者应该结合实际情况，提出自己的见解，让面试官知道你对他们公司的一种认可。

一个优秀的答案总是来自于你所作的调查研究，你可以从企业的需求方面来回答。比如，这个公司所做的工作正是你所希望参与的，而且他们做这个工作的方式极大地吸引了你，或者是企业强大的管理吸引了你，或是公司对人才的重视吸引了你。可以参考如下回答：“公司自身的高技术开发环境吸引了我”；或者“我相信我能通过自己的专业知识为公司带来效益，我对这个领域有着比较深刻的认识，而且我的学习能力和适应能力能使我个人和公司上升到一个新的台阶”；或者“我认为贵公司能够给我提供一个与众不同的发展道路”等，求职者应该表明自己的应聘原因和工作意愿，能够说出与公司产品和企业文化相关的最好。

4.2.22 你觉得我们为什么要录用你

在面试中，面试官可能会提“我们为什么要录用你”的问题。与此问题相似的还可以是“我们凭什么录用你”等，这类问题具有一定的攻击性，所以回答这种问题时，求职者应该向面试官提供证据证明自己有能力胜任该项工作，根据用人单位的需要，强调专业背景，根据工作需求叙述个人能力，而不仅仅只凭口才。通过客观的数字、具体的工作成果来辅助说明更有说服力。

能够被录用的求职者一般都基本符合工作要求、对工作有着较高的热情与兴趣、有足够的信心等。例如，可以回答“我符合贵公司的招聘条件，而且热爱编程，熟悉各类常见的编程语言，作为核心人员参与过多项实际系统的开发工作，代码量在 10 万行以上，具有较为丰富的实践经验，同时我具有高度的责任感与良好的适应能力和学习能力，完全有能力胜任该项工作，所以我非常希望能够成为贵公司的一员。”

4.2.23 你的职业规划是什么

几乎每一个求职者都不希望被问到这样的问题，但是几乎每个人都会被问到，结果回答几乎千篇一律，就是回答当管理者，然后滔滔不绝地描述自己的宏伟目标。“不想当将军的士兵不是好士兵”这句话确实有道理，想当管理者固然也没有错，但在此种场合如此直截了当地回答有时候很难得到面试官的认可，甚至招到面试官的反感。

其实，做职业规划，一定要有事实依据，而不要空想或是做白日梦，因为规划是预测未来的行动，确定将来的目标，因此需要非常详细与明确，包括何时实施、何时完成、时序安排、检查机制等。所以，回答职业规划目标的时候，也可以考虑一些别的内容。以 IT 企业为例，现今许多公司都已经建立了专门的技术途径，这些工作地位往往被称为“顾问”、“高级软件工程师”、“系统架构师”、“技术总监”等，或是进行技术划级，高级别的技术人员在待遇、地位等方面等同经理等管理层人员，所以回答此类目标一般也很合适。

当然，如果真的对某些方面有兴趣，说出也无妨，如产品经理、销售经理等一些与专业有相关背景的工作。

面试官提出此类问题的目的主要是考核求职者是否具有进取心，此时如果回答说“不知道”，也许就会使自己丧失一个好机会。如果实在不知道如何回答，最简单的方法就是回答“我准备在技术领域有所作为”或“我希望能按照公司的管理思路发展”。

4.2.24 你对薪资有什么要求

优秀的程序员可以完成的事情，10个、20个甚至更多的普通程序员都不一定能够完成，优秀与普通在IT界所能产生的价值会差别非常大，所以薪水自然也会存在着巨大的差别。但面试官提出有关薪水的问题却是一个非常敏感的问题，面试的过程中此类问题也被面试官经常提及。面试官提出此类问题，对于求职者而言确实很棘手，如果求职者要求的太低，那显然贬低了自己的能力，而如果要求太高，又会显得分量过重，公司受用不起。一些企业通常在招聘前，就对职位定下开支预算，因而他们第一次提出的价钱往往是他们所能给予的最高价钱，他们询问求职者其实也只不过想证实一下这笔钱是否足以引起求职者对该工作的兴趣。

在市场经济规律下，在企业里面，一般就是干多少活，领多少钱，所以对于此类问题，采用以下几种方式回答一般比较容易得到面试官的认可。

(1) 我对工资没有硬性要求。我相信贵公司在处理上会妥善合理。我注重的是找对工作机会，所以只要条件公平，我不会计较太多，当然包括金钱。

(2) 我对计算机编程有着系统的认识，学生阶段获得过系统分析师资格、微软认证、思科认证等，熟悉C/C++/C#/Java等多种编程语言以及精通算法与数据结构，有着多个实际的项目经验，不需要进行大量的培训就可以融入到实际的项目中去，同时我本人也对编程特别感兴趣。因此，我希望公司能根据我的实际情况以及当前市场标准的水平，给我一个合理的薪水。

(3) 对于待遇我是这样看的，作为一个应届毕业生，也是一名企业新人，想得更多的应该是能为企业作出什么贡献而不是从企业获取多少财物，自己在工作的过程中个人能力得到发展，企业也因为我的发展和贡献而受益，自然不会亏待我，这本身是一个互惠互利的过程。所以我觉得，期望配得上自己能力的薪水才是最重要的。

当面试官要求你必须自己说出具体的数目时，求职者就不要说一个宽泛的范围，否则你将只能得到最低限度的数字。最好结合当前的同类公司的市场行情以及师兄师姐对你的建议，给出一个具体的合理的数字，这样既能表明你已经对当今的人才市场作了调查，知道像自己这样学历的雇员有什么样的价值，也表明你对公司有着非常明确的期望。

4.2.25 你有什么需要问我的问题吗

面试进行到最后，面试官一般都会给求职者一个向自己提问的机会。当面试官对求职者提出此类问题时，求职者一定要有提问，而不是干脆地回答没有，因为当你回答没有的时候，面试官往往会理解你对他们公司、对这份工作没有太浓厚的兴趣，进而降低录用的可能性。如果你能够在面试官问了这句话之后提出一些问题，表现出你对公司和行业的兴趣，你的胜算将更加有保障。

其次，求职者要通过发问，更多地了解关于这次面试、这家公司、这份工作的相关信息，如企业文化、部门之间的同事情况、企业发展等，可以提问一些个人比较关注的自己所申报的职位的具体工作、发展机会、等待面试结果需要多长时间等相关的内容。

再次，就是通过向面试官提问，进一步强调自己在面试过程中没有机会谈及的个人优势，既起到了询问面试官的目的，也突出了自己未能提及的优点。例如，可以提问：“我想知道在业余时间，贵公司是否会组织一些集体活动，如篮球、乒乓球、羽毛球、排球比赛等，因为我在学校的时候，比较喜欢体育锻炼”。

通过提问，一方面，求职者可以了解到企业的一些管理理念；另一方面，也间接地暗示面试官自己在某一领域有长期发展的计划。如果准备充分的话，求职者还应当对所应聘的行业提出自

己的见解，包括现状的分析、趋势的预研等，从而提升在面试官心目中的形象，增强被录用的可能性。例如，“公司的长远目标和战略计划您能否用一两句话简要为我介绍一下？”“目前这个职位最紧要的任务是什么？如果我有幸加入贵公司，您希望我三个月完成哪些工作？”

对面试官提问也切忌提一些容易暴露自己缺点、不自信的问题。例如，“你们对毕业学校有要求吗？我不是名牌大学毕业的，你们会要吗？”、“我没有做过嵌入式开发，你们还会要我吗？”、“你们单位今年招聘几个人”等。此种问题不仅不能增加面试官对你的打分，甚至会降低在他们心中的形象。也不要揪住企业的短板一直追问，或者流露出很怀疑的态度，有些问题也不是一个普通的面试官能够回答的了的。例如，“现在云计算技术这么火，为什么贵单位不在这个方向投入人力物力了”。对于超出求职岗位或者太过高深的岗位，求职者最好还是保持警惕，超出求职岗位的问题，会让面试官觉得自己对所报岗位本身没有兴趣，过于高深的问题也会让面试官觉得自己好高骛远，引起反感。

4.3 签约这点事

其实，找工作就像找爱人一样，不存在好坏之分，只有适合与不适合，别人觉得好的，你未必觉得好，你觉得好的，别人也未必觉得好，“豆腐白菜，各有所爱”，找一份自己喜欢、适合自己的工作才是最最重要的。

4.3.1 风萧萧兮易水寒，offer 多了怎么办

工作的目的是什么？往大的方向讲，就是报效祖国、实现自身价值，往小的方向讲，就是好好生活，衣食无忧。在找工作初期，最大的愿望仅仅只是希望能够找到一份养活自己的工作而已，还没有精力来考虑其他的问题，而当有了一个 offer “保底”的时候，就会考虑发展空间、福利、节假日等问题了。每个人都希望找一份最适合自己的工作，其实找工作最大的痛苦不是一个 offer 都没有，而是 offer 太多了不知道选择哪一个，但真的需要作出选择的时候，还是需要从长计议，外企、国企还是民企，大企业还是小企业都是求职者困惑的根源。

在比较国企、外企、民企的 offer 时，并非钱越多越好，企业越大越好，其实因人而异，还需要考虑到就业的城市的生活成本、待遇（工资、奖金、福利、股票、期权、公积金、保险、过节费等）、房价、户口、未来子女入学、就医、休假情况、离家距离、发展前景、加班情况、工作压力等因素，工资是税前工资还是税后工资；股票如何分配等都需要考虑清楚。所以最终如何选择一份适合自己的 offer，还要求求职者结合自身的实际情况进行谨慎选择。

而对于大企业还是小企业的选择问题，也是因人而异。一般而言，如果现在刚刚毕业的大学生，选择大企业作为职场的开始还是更好一些，去大企业学习锻炼，不仅可以增长自己的业务能力，最重要的是还可以借助大企业的大平台快速建立人脉关系并积累资源，这些东西的重要性会随着年龄的增长越来越重要，远远超过自身的能力增长。

4.3.2 签约、违约需要注意哪些事项

经过了紧张激烈的笔试面试后，最后过五关斩六将，终于得到了用人单位的认可，拿到了用人单位的 offer，接着要做的事情就是与用人单位进行签约，以保住“胜利的果实”。在随后的求职过程中，如果遇到了感觉更加适合自己的单位，可是已经与其他单位进行了签约，只有先与已经签约的单位办理解约手续，才能与新单位进行签约，此时就涉及了违约。签约与违约是求职过程中非常常见的现象，是一种双向选择的过程，所以求职者不必为此有过大的心理

压力与负担。

(1) 签约。通常所说的签约，一般是指签订三方协议，而不是签 offer 或劳动合同。offer 只是企业对求职者的一个单方面录用意向，它不具备足够的约束力，如果只是签订了 offer，企业可以随时拒绝求职者，求职者也可以随时拒绝企业。

在签订三方协议前，首先需要弄清一个概念，什么是三方协议？三方协议不等同于劳动合同，但毕业生由签约到正式工作之间会存在一段时间差，签订三方协议就具有一定的广泛性，它是由学校、毕业生、用人单位三方共同签署后才能生效的。它也是学校制定就业方案、用人单位申请用人指标的主要依据，对签约的三方面都有一定的约束。所以，签约是一件非常严肃的事情，在签订三方协议的时候，求职者一定要认真阅读三方协议上的相关事项。

具体而言，在签约前，求职者一定要做好充分的调查工作，包括企业的规模、效益、管理制度、隶属单位、福利待遇等内容，在三方协议上一般写明与户口、薪酬、违约金相关的内容信息。需要注意的是，如果准备继续读书深造或准备出国留学，应该事先向企业说明清楚，并且写在协议书中，如果恶意隐瞒，对求职者自身和企业而言都将会带来不好的影响和不可预知的麻烦。

1) 户口。北京、上海是无数有志青年梦寐以求的地方，同时由于城市发展的限制，国家对该地区的户口管制比较严格。所以在与企业进行签约的时候，一定要区分“解决户口”、“可以解决户口”、“排队解决户口”、“抽签解决户口”、“可能解决户口”、“一般不解决户口”、“不解决户口”等各种用词之间的关系。如果真的迫切需要当地户口，一般进入国有企业（如科研设计单位、高校等）可能机会更多一些。当然，没有户口也并非无法享受医疗待遇、子女无法入学，随着国家政策的变化，未来总是朝着更加完善的方向发展。

2) 薪酬。薪酬是每一个求职者最关心的问题，也是最困惑的问题。什么才是高薪？税前工资与税后工资有什么区别？“五险一金”是什么？奖金能占工资的多大比例？

其实，作为一名求职者，在与企业谈论薪酬时，千万不要只看面上的钱，也不要只看短期内的收入，同时也不要听信 HR 说的可能的收入，而要看自己实际真正能到手的年收入，以及当地的消费水平、未来的发展前景。具体而言，薪酬主要包括以下几个方面的内容：工资、奖金、补贴、福利、股票（期权）、保险、公积金。

① 基本工资。也称为“死工资”，分为税前工资与税后工资两种，除此之外，还需要问清楚，发放工资的月数，因为很多企业每年发放工资的月数不是 12 个月。

② 奖金。很多企业的奖金占收入的很大一部分。例如，项目奖、年终奖等名目繁多的奖金。对于奖金，不同的单位情况也各不相同，奖金的数额也不一样，一般为几千至几万不等，少数企业的奖金甚至十几万、几十万，所以一定要问清楚奖金，一般 HR 们说的奖金的区限，普通求职者或者应届毕业生一般只能拿到最低数，只能取下限。

③ 补贴。有些单位会有各种补贴，一般包括通信补贴、住房补贴、交通补贴、伙食补贴等。

④ 福利。对于一些国企和事业单位来说，往往会有一些福利，而福利一般很难用金钱来衡量，如过节费、防暑降温费、取暖费、电影票、生活用品等。

⑤ 股票。对于很多企业而言，股票是一种非常有诱惑力的激励手段，但一般来说，已经上市的公司提供股票的可能性不大，反倒是一些即将上市的公司提供股票的可能性很大。所以，一定要看准机遇，走一步看三步，视野开阔，不能只贪图眼下的工资，要是提供股票的企业，即使工资比较低，也应该认真考虑。

⑥ “五险一金”。“五险一金”即养老保险、医疗保险、失业保险、人身意外伤害保险、

生育保险、住房公积金。如果求职者到私营企业、民营机构或被聘用到不占其行政编制的机关企事业单位，就需要向用人单位提出社会保险问题，至少要参加“基本养老保险”和“大病医疗保险”。

⑦ 未来收入。未来收入是指工作多年后的收入，一般而言，新员工的收入比较低，但是一旦“媳妇熬成婆”之后，收入就会大幅度提高。

所以，在三方协议中，一定要尽可能详细地将薪酬描述清楚，这样能够给予自己最大的保护。

3) 违约金。任何企业都无权不让求职者进行违约，每个企业都有追寻更优秀员工的权利，每个求职者都有追求更适合自己的企业的权利，所以违约是一项非常正常的行为。一般而言，在签约时，企业的签约负责人会主动跟求职者提及违约金是多少，然后写到三方协议的备注栏里，也有少数企业不需要违约金，只是需要等待办理违约函的时间更长一些。

签约具有一定的程序，首先求职者凭借企业的接收函到学校院系领取教育部的就业协议书，然后将该就业协议书交由企业，企业签署意见后再交给学校，最后学校签字后，此协议书生效。由于签约是各方进行交互的过程，作为弱势群体的一方，一定要认真保护自己的合法权益，当求职者将手中唯一的一份三方协议寄给企业后，此时有权利要求企业尽快签署三方协议，如过发现企业有拖延的情况，可主动与企业联系询问情况，必要时可向学院或学校就业中心寻求帮助。当求职者收到企业已签字盖章的三方协议后，应尽快确认协议书中的内容，如发现有任何问题，应及时地与企业联系。如对协议书没有异议，应尽快在协议书上签字，完成签约手续。如果条件允许，应该尽可能采取现场签约的方式以减少不必要的沟通障碍，如果因为条件限制需进行异地签约，则签约双方应保持良好的沟通和配合，尽快履行签约手续。

由于三方协议不等同于劳动合同，求职者到企业报到后，一般而言需要经过三个月到一年不等的试用期，试用期结束后，通过考核会签订正式的劳动合同，因此在签约前了解合同书的内容是十分必要的，尤其是合同书的工作年限和待遇。

(2) 违约。随着招聘工作的展开，当求职者遇到了一个他觉得更适合自己的单位的时候，由于已经与其他单位完成了签约，此时就涉及了违约。经过求职者、用人单位、学校共同通过的三方协议书具有法律效力，任何一方如果擅自解除，就被称为违约。

是否值得违约是一个很难说清的问题，因人而异，所以在进行违约前，一定要认真考虑清楚，计算一下违约成本，具体应该考虑以下几个方面的问题。

1) 新单位是否比原单位高一个档次？如果两家企业确实不在一个数量级，可以违约，但如果两家企业差不多，没有多大的区别，就没有违约的必要。

2) 新单位的签约最晚时间是什么时候？如果最晚签约时间还无法办理与原单位的违约手续、拿到学校新的三方协议，就不要冒险进行违约。如果实在无法与原单位在规定的日期内办理违约手续、拿不到新的三方协议办理签约手续，可以尝试与企业签订两方协议。

对于求职者而言，被用人单位拒绝是一件非常痛苦的事情，但是拒绝一家用人单位去选择另一家用人单位更是一件痛苦的事情，谁都不知道最终的选择是否正确。所以，对于违约，一定要谨慎，一旦违约，无论对单位、学校还是个人，都会造成巨大的伤害。同时自己可能会为自己的违约行为付出巨大的代价，如巨额的违约金。

违约的原因其实很简单，就是遇到了更好的单位。一个基本的违约流程一般包括以下 4 个步骤：

第一步，求职者需要与原签约单位进行协商，向原签约单位提出违约请求，按照三方协议规定，交纳一定的违约金（但并非所有单位都会收取违约金），从原签约单位开出解约函。

第二步，求职者从新单位获取接收函，即 offer。

第三步，将原签约单位的解约函与新单位的接收函一并带往学校就业指导中心，同时填写一份申请违约表，从学校领取新的三方协议（有时也不需要接收函）。

第四步，携带新的三方协议与新单位签约。

签约要慎重，违约更要慎重。违约不仅需要支付一定额度的违约金，同时也浪费了一个机会，对用人单位、求职者都是一种耗费精力的事情。

4.4 小结

本章针对面试中常见的一些非技术型问题，不仅分析了问题产生的背景、动机，描述了问题的回答要点，而且还给出了一些应对实例，但这些内容也并非“标准答案”。例如，当面试官问求职者：“5 年以后你希望你能获得一个什么样的职位？”一句“和你一样”的回答可能会让个性张扬的美国人非常欣赏，却可能在内敛含蓄的东方人面前败下阵来。所以，除了掌握这些常见的面试技巧外，“因人制宜、因时制宜、因地制宜”才是面试成功的关键。

英文面试攻略

第5章

Go to the United States, earning dollars!

挣美元一定要去美国吗？不用，去外企也行，越来越多的外企将重心放在了国内，在中国成立研发中心，在外企挣的是美元，花的是人民币，而且还有希望出国学习考察，与不同地域、不同肤色的人一同工作，所以很多人都选择将外企作为自己职业生涯的起点。鉴于外企工作性质的特殊需求，一般都会对求职者进行英语面试，所以英文交流能力考察成为外企的一道独特的“风景线”。

5.1 注意事项

英文面试与中文面试有很多相似之处，但由于英文语言自身的特殊性以及中英文化的差异性，使得常规的应对中文面试的方法与策略无法完全满足在英文面试中的实际需求，还需要针对英文面试的特点制定一些应对措施。

除了需要掌握一些常见的面试技巧以外，因为英文面试的特殊性，在英文面试中，还需要重点注意以下一些特别事项：

（1）描述口语化。针对面试官的提问，在用英语回答的时候，求职者不要为了说英语而说英语，不要大量使用事先准备好的花哨词汇及句式，而真正针对面试官所提问题的、与工作有关的个人见解却很少，内容空泛，逻辑混乱。最后可能得到一句英语不错的夸奖，除此之外，什么都没有了，自然也不会被录用。所以，作为求职者，要明白一个道理，用简单直白的语言表现最具魅力的自我，才是英文面试的最高境界。有些人习惯于堆砌华丽辞藻，明明可以简单描述的内容非要用多种从句加以限定，恨不得在面试中用大气磅礴的英文演讲征服面试官，犯了舍本逐末的错误，所以描述要口语化。

口语的特点在于结构的不完整性和与说话场合的紧密依存性，口语中多使用“and”、“but”之类的连接词，并且需要多使用单一的动词结构，而少用复杂的并列句或从句。同时，在用英语回答问题时，切忌语速过快，有些求职者以为只要在外国面试官面前把英文说流利即可，越流利越能说明自己水平高，其他的无所谓，于是语速特别快。其实这是不对的，如果语速太快，老外根本就听不懂这种赶火车似的中式英文。英文面试的目的虽然也是为了考查英语口语表达能力，但不是炫英语口语，而是让外国面试官了解你，甚至对你感兴趣，最重要的是展示你的综合素质，绝非单单的语言能力，表达清晰是首位，流利是第二位。而且，语速过快也容易给人不自信、不稳重的错觉。

（2）时态的变化应用。由于语言自身的因素，不同于中文表达，在英语表述中需要注意时态的变化应用，当面试内容涉及个人经历、教育背景、工作经验、未来规划等问题时尤其重要，所以一定要使用正确的时态，否则面试官很可能弄不清求职者描述的内容是过去、现在或是未来将要发生的，影响其对求职者的认识，最终影响录用结果。

（3）文化差异。由于面试带有一定程度的主观性，面试官的喜好会很大程度地决定求职者是否能够获得企业的青睐，而面试官很有可能来自不同的国家与地区，有一定的个人倾向，所

以在面试过程中，应该尽可能避免一些引起面试官不高兴或反感的问题。针对此种问题，首先要避免使用过于生僻的英文单词或是地方俚语等接受群体相对较小的表达方式，这些表达方式有可能会造成听者的困惑与曲解。其次就是要避免过多、过于主观地谈及宗教文化或时事政治方面的问题。所以在求职的过程中，在不了解情况的状态下，如果谈到一些敏感话题，谨慎而有节制的发言才是上策。

(4) 以英语展示才能。对于非英语专业要求的工作，面试基本都是英语口语形式，而英文面试与英语考试的口试不同，面试官通常是由公司的人事主管、应聘部门主管或公司高层组成的，所以他们更关心的是求职者的专业知识和工作能力，而对他们而言英语此时只是一种交流工具，求职者具备了基本的交流能力即可，而英语交流能力也只是求职者要展示的众多技能中的一种，并非唯一，并非最重要的。所以在面试过程中，求职者切忌为了说英语而说英语，咬文嚼字、避重就轻都是不可取的，求职者应该针对面试官提出的与工作有关的问题，形成个人见解，简单明确地展示给面试官。同时回答问题的时候切忌内容空泛、逻辑混乱，否则最后除了得到一句英语不错的夸奖之外，恐怕很难有理想的收获。

面试中的英文不在于求职者口语发音有多纯正、多接近老外的发音，关键在于能与面试官无障碍沟通。你能听懂别人的话、别人能听懂你的表达才是最重要的，因为在流利的英语和聪明的头脑之间，很多人都会选择后者的，更何况面试官们。

5.2 英文自我介绍

面试官可能会提出以下问题让求职者进行自我介绍：

- 1) Introduce yourself, please. (请介绍一下你自己。)
- 2) Please let us know something about you. (请告诉我们一些关于你的事情。)
- 3) Could you please introduce yourself? (能介绍一下你自己吗?)
- 4) Could you tell me something about yourself? (可以说一下关于你的一些事情吗?)

针对面试官提出的进行自我介绍的问题，求职者应该如实回答，具体而言，体现在以下几个方面的内容：

(1) 基本信息。基本信息包括姓名、年龄、家庭等。

例句：My name is ××× (I'm ×××), I'm twenty-three years old now. There are four members in my family, my father, my mother, my brother and I. (我叫×××，今年23岁。我们家一共4口人，爸爸、妈妈、弟弟和我。)

(2) 个人性格。个人性格主要包括性格与兴趣爱好方面的内容。

面试官可能会提问：Describe your personality? (说一下你的性格?)、What do you do when you are free? (当你闲暇时你一般做什么?) 等问题，回答该类问题尽量让面试官觉得你是一个自信的人，是一个有着生活情趣的人。

例如：I am confident, well-organized, and easy to get along with. I have a lot of friends, and they trust me. (我认为我是一个自信的人。我做事很有条理，并尽力使别人认为我很容易相处。我有很多朋友，他们都很信任我。)

I'm a very passionate person and patient as well. I can take difficult jobs that bother other people and work at them slowly until they get done. Sometimes, I may look dull. But when you get to know me better, you'll see that I'm pretty smart in fact. One of my weaknesses is that I'm not good talking in front of people. I need to work on that. (我是一个热情且做事很有耐心的人。我能耐心地做一

些别人不能做的困难的事情，并能坚持到底。有时我可能看起来并不聪明，但如果你们逐渐了解我之后，会发现我是一个相当聪慧的人。我的一个缺点是在人前不太善于表达自己，我需要对此不断改正。）

(3) 教育背景。面试官可能会提出 Tell us something about your education background (说一下你的教育背景) 或者 Could you tell us something about your education background (请说一下你的教育背景吧) 的问题。提出此类问题，面试官主要是希望求职者介绍一下自己的毕业学校及专业。

所以回答这种问题，也应该简单明了。例如，I graduated from ×××. (我毕业于×××。) I learned mechanics when I was in ×××. (我在×××学习机械。)

(4) 工作经验。工作经验主要是介绍自己的工作单位、工作时间、工种及一些具体的内容。

例如：I began to work in ××× as a lathe operator from 2000. I work very hard and I can do my work very well, I hope I can have the chance to work in Singapore. (我2000年开始在×××从事车床操作。我工作刻苦，能很好地完成任务。我希望我能在新加坡工作。)

自我介绍范文如下：

I'm so glad that your company gives me this opportunity for the interview today. My name is ×××, ×× years old now. ×× years ago I came to ××× university and chose ××× as my major with a strong interest in it. When I read your recruitment advertisements, I know this job is what I have been expecting.

During my college life, I spent most of my time in professional courses. I have acquired enough basic knowledge of ×××, which can be reflected by my transcript. But I don't think scores can represent everything. So in my spare time I participated in various practical activities, such as ××× and ×××. Although I lack work experience, I have paid great attention to cultivate and improve my abilities to learn and to work under pressure in my studenthood, in order to become a good learner and a good team player. In my mind a good occupational ethics quality is also valued. People who are familiar with me judge me as a reliable, lively and considerate person and they enjoy working with me. I have confidence that in the future I will become a member of your company.

Hope my introduction can impress you. If you feel I am suitable for this job, please give me a chance. Finally, thank you very much for giving me this precious interview. Wish you all a good day.

对应的中文意思如下：

我很高兴能有机会来参加贵公司今天的面试。我叫×××，今年××岁。××年前，因为对×××有着巨大的兴趣，我选择进入×××大学学习×××专业。当我看到这则招聘信息的时候，我发现这就是我一直期待的工作。

在大学里，我将主要精力放在了专业课的学习上，并且掌握了良好的基础知识，这可以通过我的成绩单反映出来。但我并不认为分数能够代表一切，所以业余时间我参加了各种社会实践活动，如×××和×××，尽管我缺少工作经验，但是在我的学生生涯中，我特别注重培养和提高自己学习能力和抗压能力，以成为一名优秀的学习者和团队合作者。我认为好的职业道德素养也是非常重要的。熟悉我的人认为我是一个可靠、活泼、考虑周到的人，并且他们喜欢和我一起工作，我有信心在将来成为贵公司的一员。

希望我的自我介绍能够打动各位，如果各位觉得我适合这项工作，请给我一个机会。最后很感谢能给我这样一个宝贵的面试机会。祝各位愉快！

好的礼仪是一个好的面试开始，在自我介绍之前一定要注意基本的礼仪，以下是常用的问候语与开场白。

- 1) Good morning, everyone. (大家早上好。)
- 2) It's my great honor to introduce myself to you here. (很荣幸在这儿向大家作自我介绍。)
- 3) It's my pleasure to introduce myself to you here. (我很高兴在这儿向大家做自我介绍。)
- 4) I'm very happy to introduce myself to you here. (我很高兴在这儿向大家做自我介绍。)
- 5) It is a privilege to be speaking to you today. (今天能跟您谈话真是三生有幸。)

5.3 常见的英文面试问题

对于很多英文学习者而言，其目的与动机其实很单一，就是为了应试，而希望在外企工作或出国的人，则需要从心底里热爱英语，学习英语，提高自己的英文水平。以下是一些常见的英文面试问题以及对应的推荐回答。

(1) What do you know about our company?

Answer: Your company is the largest online marketing firm in the industry. This year analysts expect it to generate a record turnover of over \$10 billion. It is said that the employee benefits of your company are very good. According to the schoolfellows' speaking, the working environment in your company is friendly, and so as to the working atmosphere. The key reason why I choose your company is your focus on developing new staff. I desire to work in a company like this.

对应中文翻译

问：你对我们公司有什么认识？

答：贵公司是业界最大的在线营销公司。最近分析人士预计它将创造出 100 亿元的营业额，员工福利据说也不错。听在贵公司工作的校友讲，公司的工作环境很好，工作氛围比较宽松，而且十分注重新人的培养，这是我选择贵公司的主要原因。我很希望到这样的公司工作。

(2) I think one can adapt to a new job quickly if he/she has some work experience. But you are a fresh graduate according to your resume. How are you going to deal with this challenge at work?

Answer: I admit that work experience may be helpfull at times. But through more than 10 years' studenthood, I have developed a good learning ability and working initiative. In my college time, I learned basic lessons, and at the same time I taught myself C# and Java. I have cooperated with my classmates to complete several subjects. In a sense, being inexperienced in work can be my advantage, because I will have more strong feelings of freshness and desire to learn. I am not afraid of pains or hardship. I believe I can adapt to the new job as soon as possible with a steady working attitude and a hardworking spirit.

对应中文翻译

问：我们认为有一定的工作经验能更快地适应本职位，但是通过你的资料看来，你是一名大学应届生，你认为你应该如何应对这方面的不足？

答：我承认工作经验有时具有一定的帮助，但是经过在学校的培养，我已经养成了良好的学习能力和工作主动性。在校期间，我在学好基础课的同时，自学了 C# 和 Java，并且和同学一起完成了几个较大的项目。同时，在某种意义上，我认为没有经验的我对这份工作有更高的新鲜感和学习欲望；而且我不怕付出更多的努力和汗水。我相信有踏实的态度和好学的精神，我会尽快地融入工作。

(3) What are your salary expectations?

Answer: The normal salary in this company for this type of work ranges from 5000yuan to 7000yuan, which is fairly reasonable for me. But at present I don't quite understand your company's wage levels, maybe I have made an inaccurate evaluation. Meanwhile, I think the salary is a representation of one's work capacity. I hope my wage can have some change based on my job performance after I am familiar with the business processes. I am wondering whether you could tell me what is the suitable salary range in your mind.

对应中文翻译

问：你希望这个职位的薪水是多少？

答：我了解到这份工作的薪水的范围大概是 5000~7000 元，这对我来说很合适。但是由于我现在还不是很了解贵公司的薪资水平，可能给出了错误的估计。同时我认为工资是一个人的工作能力的体现，我希望在熟悉工作流程后能够根据我的工作表现适当地改变我的薪资。不知道您能否告诉我这份工作在您心目中的合理工资范围是多少？

(4) How do you think about your failures in life?

Answer: Failure is an essential part of life. None of us was born "perfect". Failure can make us more awake and realize our weaknesses, and bring new energy and experience to our life. I think if one thing is worth doing, it is worth doing well. Nevertheless, we should work pulling out all the stops. Put our foot down and we can prevent the needless failure.

对应中文翻译

问：你如何看待自己生活中的失败？

答：我认为失败是生活必不可少的一部分，我们大家生来都不是十全十美的。失败有时能让我们更加清醒，从而意识到自己的不足，为我们的生活注入新的能量和经验。我认为如果一件事值得去做，就应该把它做好不过，我觉得面对生活要全力以赴，脚踏实地，以避免不必要的失败。

(5) What do you like the most about your college courses?

Answer: I prefer some professional courses, such as Date Structure, Algorithms and so on, because of their combination of theories and practices. Every time I write the right codes using the knowledge on the books, I feel a bit of achievement. Besides, I also like the PE classes. Our PE teacher is a man full of humor who always makes the students laugh while work out.

对应中文翻译

问：你最喜欢的大学课程是什么？

答：我比较喜欢一些专业课，如数据结构、算法等，因为它们是理论和实践相结合的。当自己能够把学到的理论用正确的代码运行出来时，我会觉得有小小的成就感。此外体育课也是我比较喜欢的课程，因为我们的体育老师很幽默，我们能在欢笑的同时还锻炼了身体。

(6) Then how do you feel about your team spirit?

Answer: Honestly, my teamwork spirit is relatively good. We had a lot of large projects in college. Every time I could complete my mission successfully. If admitted, I can show you the air-booking system and explain my project part. I believe the saying that two heads are better than one. In face of a big project, teamwork can achieve the goal fully using each other's advantages, and raise the work efficiency at the same time.

对应中文翻译

问：你认为你的团队合作精神怎么样？

答：我自认为自己的团队精神相对来讲还不错。因为在大学期间，我们有许多大的项目，每次我都能出色地完成任务。如果允许，我可以向您展示我们合作完成的航空订票系统，并说明我的工作部分。我相信三个臭皮匠胜过诸葛亮这句谚语。在遇到大项目时，团队合作可以实现成员间的优势互补，同时能够提高工作效率。

(7) Why did you choose your college major?

Answer: I chose Software Engineering as my major because of its development potential and good prospects for employment. At that time, our government had introduced a series of policies to encourage the development of the software enterprises. I also got the news that Qingdao was building a ×× Software Park. I could just right apply for a job there when I would graduate from university. And ×× University's Computer Science major was started earlier and its graduates are competitive in job market. I am good at English and math. So I chose the major of Software Engineering.

对应中文翻译

问：你为什么选择了现在的专业？

答：我当初选择这个专业是看重了软件工程良好的发展潜力和就业前景。在我选择专业时，国家正好出台了一系列政策，大力扶持软件企业的发展，我还从新闻上得知青岛正在兴建 ×× 软件园，当我毕业时正好可以来这里工作。而且 ×× 大学的计算机专业起步较早，毕业生在就业市场上也具有竞争优势。而且我的英语和数学成绩较好，所以我选择了软件工程专业。

(8) Can you talk something about what you've learned from your internship?

Answer: During my internship, I got familiar with the work flow preliminarily and experienced the highly disciplined enterprise culture. By joining in the ×× project, I learned how to use the ×× software tools. And I know that the basic knowledge in books is very useful for practice. As I've forgotten some of the theories, I am reviewing some essential knowledge these days. During the time in the company, I deeply felt that there was a big difference in atmosphere between corporations and schools. I still need time and study to complete the change mentally.

对应中文翻译

问：能说说你在实习期间的收获吗？

答：在实习过程中，我初步熟悉了公司的业务流程，感受到了严谨的企业文化。通过参加 ×× 项目，我学会了运用 ×× 工具。通过实习我也发现，书本的基础知识对于指导现实实践是很有用的，而一些课程的理论我已经忘掉不少，所以这段时间，我一直在巩固基础知识。在公司的这段日子里，我深刻感受到企业氛围和学校有很大的不同，我还需要时间和学习来完成心态上的转变。

(9) How did you get our recruitment information?

Answer: I read your recruitment information on the college recruitment site. Then I sent you an E-mail with my resume attached and finally I got your interview message.

对应中文翻译

问：你是从哪得知我们的招聘信息的呢？

答：我从学校的招聘网站上得知了贵公司的招聘信息，并给贵公司的招聘邮箱发了一份简历，收到了面试通知。

(10) What kind of workplace spirit do you value most?

Answer: As a business man, practical mind, innovative spirit, the overall situation consciousness, sense of service, eagerness to learn and good teamwork spirit should be highly valued. But most of us cannot do it all.

对应中文翻译

问：你崇尚的职场精神是什么？

答：我认为作为一个职场人，应该具有实践精神、创新精神、全局意识、服务意识、学习意识和团结合作的素质。但是我觉得大多数人可能很难做到面面俱到。

(11) If the company has a requirement for a business trip, do you mind that?

Answer: I don't mind at all. Now I don't have the burden of my family. So I will be able to concentrate more on business during the business trip. And I like traveling around when I am young. I think business trips are an enjoyable part of the work.

对应中文翻译

问：如果公司有出差要求，你介意吗？

答：一点都不介意。我现在还没有家庭负担，出差时会有更多的精力放在业务上。我喜欢趁着年轻的时候到处转转，所以我觉得出差是工作中很享受的部分。

(12) We may ask some employees to work overtime occasionally. What's your attitude about working overtime?

Answer: I have heard that the IT enterprises may request the employees to work overtime and I have made enough preparations. If necessary, I guarantee that I will be duty-bound. At the same time, I will raise my work efficiency to reduce overtime hours.

对应中文翻译

问：公司有的时候可能有加班要求。你是怎么看待加班的？

答：我已经听说过 IT 企业的加班要求，并且已经做好了充分准备。如果工作需要，我会义不容辞地加班，同时我也会提高自己的工作效率，减少不必要的加班。

(13) Can you give us a brief introduction?

Answer: Certainly. It is really a great honor to have this opportunity for an interview. My name is ×××, 23 years old now. I graduated from ××× University, and my major was computer science. Through four years' hard working, I have mastered the professional courses, such as C++, Data Structure, Computer Network, etc and I have passed the CET6 examination. I have ever participated in your company's internship. I really hope I would get a chance to work for your business.

对应中文翻译

问：你能简单地介绍一下自己吗？

答：当然可以。很荣幸有机会参加今天的面试。我叫×××，今年 23 岁。毕业于×××大学计算机专业。通过 4 年的学习，我已经熟练掌握了 C++、数据结构、计算机网络等专业知识，通过了 CET6 考试，并且参加了贵公司的企业实习，希望有机会来贵公司工作。

(14) What are your good qualities?

Answer: I have developed my excellent abilities to learn and to work under pressure in order to become a good team player in my studenthood. People who are familiar with me judge me as a reliable, sunny and considerate person and they enjoy working with me. Besides, I have strong organizational skills, and I ever participated in organizing Welcome Parties and New Year's evening performances.

对应中文翻译

问：你有什么优点？

答：在大学期间，我培养了良好的学习能力和受压能力，是一个好的团队合作者。熟悉我的人认为我是一个可靠、开朗、为别人着想的人，而且喜欢和我一起工作。同时，我还有良好的组织能力，曾经参与组织学院的迎新晚会和新年晚会。

(15) What do you consider your weaknesses?

Answer: My weakness is that it is easy for me to get bored by routine. By comparison I prefer something fresh. I can't bear procrastination. If a task is given, I hope it can be settled as soon as possible. Too much emphasis placed on this work may obscure others. Ever I didn't like writing, so I put an emphasis on my document ability in my internship and now I think I have had a big progress. Anyway, I will be extremely receptive to others' suggestions and take initiative to improve myself.

对应中文翻译

问：你认为你有什么缺点？

答：我的缺点是容易对日常的循规蹈矩感到厌烦，相比之下我更愿意尝试一些新鲜事物。我不能忍受拖延，如果一份工作任务下达之后，我希望尽快完成，可能因此会忽略其他事情。曾经我不喜欢写作，但是经过大学期间的实习，我着重锻炼了自己的写作能力，现在已经有很大提高。总之，我乐于接受别人的意见也会主动去改正。

(16) Would you mind talking about your family?

Answer: Certainly not. I was born in a family of ordinary workers. My parents have paid extreme attention to my grow-up, and they invest a lot in me. I choose this job since they are now living in this city. I hope I can take care of them frequently.

对应中文翻译

问：介不介意谈谈你的家庭？

答：当然不介意。我出生于一个普通的工人之家，但是我父母十分重视对我的培养，为我付出了很多心血。他们就生活在这个城市，所以我选择了这份离家较近的工作，希望可以多照顾他们一些。

(17) Can you tell me something about your hobbies?

Answer: I like music, and in my leisure time I often go to the KTV with my friends. When the weather is fine, I may go climbing. I think these hobbies exercise my perseverance and help me release the pressure.

对应中文翻译

问：你有什么爱好吗？

答：我喜欢音乐，闲暇时经常和朋友去 K 歌；天气好的时候，会去爬爬山，我认为这些爱好锻炼了我的毅力，也帮助我释放了压力。

(18) What was the reason that promoted you to leave your current company?

Answer: I love my last job. But I feel I have reached the peak level at the last company. There is no good opportunity for advancement. Your company gives me a better chance. And I am ready for more challenges and responsibilities. Now that the opportunity is coming to me, I will take it.

对应中文翻译

问：什么原因促使你离开现在的公司呢？

答：我很喜欢那份工作，但是对我来说在上一个公司已经到达顶峰，没有进步的机会，而

你们给了我一个更好的机会，我准备好了面临更多的挑战和职责，既然机会来了我认为我应该抓住它。

(19) What do you do with people's criticism?

Answer: I will accept it firstly and then rethink it in patience. If the criticism just points out my shortage, I will consult with him humbly. While as for the spiteful remarks, I will just forget it.

对应中文翻译

问：你通常如何处理别人的批评？

答：我先会理性地接受，然后耐心地反思。如果他的批评的确指出了我的问题所在，我会虚心向他请教；如果是恶意的批评，我也不会让它影响我的心情。

(20) What kind of people do you like to work with?

Answer: I'm an easygoing person and usually don't demand much of others. As a new worker, I wish I could get familiar with the environment, change my studenthood mentality and display my special skills quickly.

对应中文翻译

问：你喜欢什么样的工作伙伴？

答：我是一个很容易相处的人，对别人要求不是很高。作为一个新人，我希望我能尽快地融入工作环境，改变学生时代的心态，尽快发挥出我的专长。

(21) What kind of predictable difficulties do you think about this work?

Answer: It is normal and inevitable that there exist some difficulties at the beginning. Especially for the new workers, new circumstances and new work flows are the first two challenges. But in the process of accepting the challenges, surmounting the difficulties and enduring the hardship, success can reflect its value. As long as there is indomitable perseverance, a challenging spirit and a good team work spirit, any difficulty can be overcome.

对应中文翻译

问：对这项工作，你有哪些可预见的困难？

答：我认为工作开始时出现一些困难是很正常且难免的。尤其是对于一个新员工来说，新环境和崭新的工作流程都是首先要面对的两个挑战。但是在接受挑战、克服困难、忍受磨难这个过程中，成功才会体现出它的价值。只要有坚韧不拔的毅力，敢于挑战的精神，良好的团队合作意识，任何困难都是可以克服的。

(22) If we hire you, how long will you work for us?

Answer: I will work here as long as possible and strictly comply with the contract requirement. After all, changing jobs is also not good for an employee's development.

对应中文翻译

问：你能为我们公司工作多久？

答：我会尽可能久地为贵公司效劳，并且严格按照合约做好自己的工作。毕竟换工作对一个员工的发展也是不利的。

(23) What do you expect from this work?

Answer: This is a big opportunity for me. On one hand, the work place is relatively close to my home. I don't have to suffer from a long-distance run and I could spend more time in taking care of the family. On the other hand, the attractive salary and welfare benefits can help me change my life economically. And above all, this job satisfies my major and hobby. It is advantageous for my career

development.

对应中文翻译

问：你希望从这份工作中得到什么？

答：这份工作对我来说是一次机会。一方面，工作地点离我家比较近，我可以免受奔波之苦，还可以照顾我的家庭；另一方面，良好的薪资福利待遇可以改善我的生活状况。最重要的是，这份工作符合我的专业和爱好，更有利于我的职业发展。

(24) Do you have a clear life plan?

Answer: Yes. I think one can reach his goal easier if he has a clear life plan. What I should do in this phase is working hard, learning about the business process, and adapting to the company culture. Once I gain sufficient experience, I would like to move on from a technical position to a management position.

对应中文翻译

问：你有没有一个明确的人生规划？

答：有。因为我认为有明确规划的人更容易成功。我现阶段的职业规划就是先好好工作，熟悉公司流程，适应公司文化。一旦我有了足够的经验，我想从技术职位转到管理方面。

(25) If we give you an offer, but you will find that you don't fit it at all some time later. What will you do?

Answer: First, I will make a serious self-question to check if I do my best. I will try to learn more from the leaders and colleagues about the business knowledge and the way to deal with daily affairs and make some changes if possible. If I eventually find I am not qualified for the job, changing a job may be the best choice, which is good for both the company and me.

对应中文翻译

问：如果通过这次面试我们录用了你，但是工作一段时间后你发现自己根本不适合这个职位，你会怎么办？

答：首先，我会从自身找找问题，看看自己是否尽了全力。平时尽量多向领导和同事学习业务知识和处事经验，然后尝试着去做出一些改变。如果最后发现自己根本胜任不了这份工作，换一份工作或许是最好的选择，这样对公司和我都有好处。

(26) What a leader would you like to have?

Answer: "Top leaders" who possess the absolute powers play a key role in commanding the overall situation of a region or a unit. As a worker, I will wholeheartedly cooperate with their decisions. Leaders, like my teachers in my mind, may be strict sometimes. But they can point out my errors in time, helping me go forward.

对应中文翻译

问：你希望有一个怎样的领导？

答：我认为领导是统领全局的关键人物，作为一个职员，需要配合领导的决策。我希望我的领导更像是我的一位老师，或许有时很严厉，但是总能够指出我工作中的错误，帮助我前进。

(27) What sort of people do you think is difficult to work with?

Answer: Every person who is difficult to work with is my potential friend. It takes time to learn the work partners' personality and rhythm. Sometimes we should be patient. Time can settle everything.

对应中文翻译

问：你认为工作中什么样的人最难相处？

答：我认为每个难相处的人都是潜在的朋友。工作伙伴需要磨合才能适应彼此的性格和节奏，有时我们应该忍让，时间会解决一切的。

(28) Do you ever want to start your own business?

Answer: Being the boss of your own company is a beautiful dream. But starting a business requires an economic base and lots of experience. I am so young that there are a lot of things to learn. It is the right time for starting to learn something. I will consider setting up my own business when I get mature.

对应中文翻译

问：你想过创业吗？

答：做自己的老板是个美好的梦想。但是创业需要一定的经济基础和业务经验，我现在还年轻，需要学习的东西很多，现在是学知识的好时机。如果想创业等自己成熟了再考虑也不迟。

(29) If you fail to get this offer, what will you do?

Answer: Failure is an essential part of life. I might be a bit disappointed for some days. But I believe life is full of opportunities. As long as you make adequate preparations, there are better chances waiting for you.

对应中文翻译

问：如果你竞聘这个职位失败怎么办？

答：失败本来就是生活必不可少的一部分，我可能会心情低落几天。但是我相信生活中充满了机会，只要准备充分，会有更好的机遇在前面等着我。

(30) Do you talk with other companies at the same time?

Answer: Yes. Because I want to find the job which suits me best and is qualified for me. It is nearly impossible to find a suitable job without trying a bunch of options at the beginning of one's career.

对应中文翻译

问：你现在有没有同时和其他公司洽谈？

答：是的，因为我想找到最适合我和我能胜任的职位。一个人开始职场生涯时，如果不尝试多种选择，不容易找到适合自己的工作。

(31) Which management style do you prefer, severe or light?

Answer: The management style depends on the company culture. Employees should learn to adapt to it. Some pressure may encourage employees to acclimatize themselves to the new environment at the start. When they have adapted to the usual business life, the pressure could decrease a little bit.

对应中文翻译

问：你更倾向于哪种管理方式，严厉些的还是宽松些的？

答：管理方式应该是因企业文化而异的，员工要学会适应企业的文化氛围。开始时严厉一些能够给员工一定的压力尽快适应企业，熟悉之后可以稍微宽松些。

(32) Do you think yourself is a good student in college?

Answer: Maybe everyone has a different standard. The answer I give to myself is "yes". I have a solid basic professional knowledge. My scores have been consistently ranked in the top 10 percent of all the students in my grade. At the same time I am a member of the Student News Agency, the class cadres, and I take an active part in the school activities, which develops my good teamwork spirit. This summer holiday I joined the practice activity for senior students in × × Company, which greatly

improved my ability of operation.

对应中文翻译

问：你认为你在学校是个好学生吗？

答：可能大家评判好学生的标准不尽相同。我给自己的答案是“是的”。我在学校时的成绩在年级前 10%，有着扎实的专业知识功底；同时担任院大学生通讯社记者、班级干部，积极参加集体活动，培养了良好的团队合作精神。今年暑假我参加了××公司为高年级学生提供的暑期实习，动手能力也有很大提高。

(33) What's your idea about education background and working ability?

Answer: The education background is not proportional to the working ability. Working ability reflects one's ability to learn and understand new ideas and new knowledge. However, education background represents one's cultural enrichment. It is rather a proof of one's education degree.

对应中文翻译

问：你怎样看待教育背景和工作能力？

答：我认为教育背景和工作能力是不成正比的。工作能力是一个人学习并掌握新事物和新知识的能力，而学历代表文化修养的程度，是一个人接受教育程度的凭证。

(34) What provide you with a sense of accomplishment?

Answer: Every tiny bit of progress will give me a sense of personal achievement, especially completing a project to the limit of my limit.

对应中文翻译

问：什么会让你有成就感？

答：我觉得生活中的每一点进步都让我感到有成就感，尤其是达到自己的极限完成一个项目时。

(35) How do you handle your conflict with your colleagues in your work?

Answer: Firstly, I will try to propose my ideas in a more courteous manner in order to get my points understood and meanwhile avoid quarrels. When we cool off, we discuss it later.

对应中文翻译

问：你怎样处理和同事的不同意见？

答：首先我会用比较谦恭的态度提出我的看法，使对方了解我的观点，同时避免争吵。等大家冷静下来再讨论。

(36) Are you a multi-tasked individual?

Answer: I think I can. I joined in three projects at the same time and accomplished my part excellently. So if the tasks are within my powers, I think I can handle them well.

对应中文翻译

问：你认为你可以同时承担多项工作吗？

答：我认为我是可以的。在研究生期间，我曾经同时参加过 3 个项目，并且出色地完成了任务。所以，如果工作是在我的能力范围内，我想我可以掌控好的。

(37) What will you do if you fulfill your work part before your workmates?

Answer: I will try my best to help them. As the saying goes, one swallow does not make a summer. Only if all members finish their tasks, can we say that the project has been successfully completed.

对应中文翻译

问：如果你先于同事完成了任务，你会怎么办？

答：我会尽力帮助同事。正如谚语所说，孤燕不成夏。只有所有人的任务完成了，我们才可以说任务成功完成。

(38) Which one is more important to you, salary or work?

Answer: If I can't have fish and bear's paw at the same time, I will choose work. Doing a job that is a bad fit can be a demoralizing experience. Only choosing the right work, can you have a good development. Then there is no doubt you will have a satisfied wage.

对应中文翻译

问：薪水和工作，哪个对你更重要？

答：如果鱼和熊掌不可兼得的话，我觉得工作对我比较重要。干一份不适合的工作会是一次令人消磨斗志的经历。只有选择了适合自己的工作，才能有更好地发展，薪水待遇也会毫无疑问地令你满意。

(39) Will you accept the department exchange if you were admitted?

Answer: I would like to agree with any reasonable arrangement by the company. But I think I am better at software testing.

对应中文翻译

问：如果你被录用了，是否会接受部门的调剂？

答：我很乐意接受公司一切合理的安排，但是我认为我更擅长软件测试方面的工作。

5.4 常见计算机专业词汇

作为计算机相关专业的学生，面试或者笔试时不可避免地会遇到与专业相关的问题，而考核专业问题的时候，又不可避免地涉及很多专业词汇，这就需要求职者掌握好常见的专业词汇。只有这样，才能在阐述问题时得心应手，避免出现表达错误引起误解。以下是计算机专业常见的相关词汇。

5.4.1 计算机专业相关课程

| | |
|------------|---|
| 计算机导论 | Introduction to Computer Science |
| 高等数学 | Advanced Mathematics |
| 应用创造学 | Creativity Methodology |
| 工程图学与计算机绘图 | Engineering Graphics and Computer Graphics Drawings |
| 面向对象程序设计 | Object-oriented Programming |
| 概率论与数理统计 | Probability Theory and Statistics |
| 离散数学 | Discrete Mathematics |
| 软件工程概论 | Introduction to Software Engineering |
| 数据结构 | Data Structures |
| 计算机组成与结构 | Computer Organization and Architecture |
| 操作系统 | Operating System |
| 计算机网络 | Computer Network |
| 算法设计与分析 | Algorithm Design and Analysis |
| 软件工程经济学 | Software Engineering Economics |
| Java 技术 | Java Technology |

| | |
|-----------|---|
| UML 建模 | UML Modeling (Unified Modeling Language Modeling) |
| 数据库系统概论 | Introduction to Database Systems |
| 编译原理 | Principle of Compiler |
| 软件体系结构 | Software Architecture |
| 程序分析 | Program Analysis |
| 软件过程与项目管理 | Software Process and Project Management |
| 系统分析与设计 | System Analysis and Design |
| 程序测试 | Program Testing |
| 模式识别 | Pattern Recognition |
| 嵌入式程序设计 | Embedded Programming |
| 人机交互技术 | Human-computer Interaction Technology |
| 云计算 | Cloud Computing |
| 计算机与网络安全 | Computer and Network Security |
| 计算机图形学 | Computer Graphics |
| 数据挖掘技术 | Data Mining Technology |
| 分布对象技术 | Distributed Object Technology |
| 网络多媒体 | Internet Multimedia |
| 网络程序设计 | Network Programming |
| .NET 程序设计 | .NET Programming Design |
| 协议工程 | Protocol Engineering |

5.4.2 操作系统相关术语

| | |
|-----------|--|
| 虚拟机 | Virtual Machine |
| 访问控制列表 | Access Control List |
| 线程 | Thread |
| 多线程 | Multithreading |
| 进程 | Process |
| 守护进程 | Daemon |
| 进程间通信 | IPC (Interprocess Communication) |
| 死锁 | Deadlock |
| 银行家算法 | Banker's algorithm |
| 内存管理 | Memory management |
| 虚拟地址 | Virtual address |
| 物理地址 | Physical address |
| 引导盘 | Boot Disk |
| 页面失效 | Page Fault |
| 后台进程/前台进程 | Background Process /Foreground Process |
| 文件传送协议 | FTP (File Transfer Protocol) |
| 图形用户界面 | GUI (Graphical User Interface) |
| 权限 | Permission |
| 移植 | Port/Ported/Porting |

| | |
|---------|-------------------------------------|
| 可移植系统接口 | Portable Operating System Interface |
| 分时 | Time-sharing |
| 工作区 | Workspace |
| 工作目录 | Working Directory |
| 窗口管理器 | Window Manager |
| 封装器 | Wrapper |

5.4.3 算法相关术语

| | |
|----------------|--------------------------------------|
| 字典 | Dictionary |
| 堆 | Heap |
| 优先级队列 | Priority queue |
| 矩阵乘法 | Matrix multiplication |
| 贪心算法 | Greedy algorithm |
| 上界/下界 | Upper bound / Lower bound |
| 最好情况/最坏情况/平均情况 | Best case /Worst Case/ Average case |
| 插入排序 | Insertion sort |
| 合并排序 | Merge sort |
| 堆排序 | Heap sort |
| 快速排序 | Quick sort |
| 动态规划 | DP (Dynamic Programming) |
| 背包问题 | Knapsack problem |
| 霍夫曼编码 | Huffman Coding |
| 迪杰斯特拉算法 | Dijkstra's algorithm |
| 贝尔曼-福德算法 | Bellman-Ford algorithm |
| 弗洛伊德算法 | Floyd-Warshall algorithm |
| 回溯 | Back-Tracking |
| N 皇后问题 | N-Queen problem |
| 渐进增长 | Asymptotic growth |
| 线性规划 | Linear programming |
| 随机数生成 | Random number generation |
| 图的生成 | Generating graphs |
| 图论-多项式算法 | Graph Problems-polynomial algorithm |
| 连通分支 | Connected components |
| 最小生成树 | Minimum Spanning Tree |
| 最短路径 | Shortest path |
| NP 问题 | Non-Deterministic Polynomial problem |
| 旅行商问题 | Traveling salesman problem |
| 同构 | Graph isomorphism |
| 压缩 | Text compression |
| 最长公共子串 | Longest Common Substring |
| 最短公共父串 | Shortest Common Superstring |

收敛速度

Rate of convergence

5.4.4 数据结构相关术语

| | |
|----------------|--|
| 集合 | Set Data Structures |
| 线性方程组 | Linear Equations |
| 数据抽象 | Data abstraction |
| 数据元素 | Data element |
| 数据对象 | Data object |
| 数据类型 | Data type |
| 逻辑结构 | Logical structure |
| 物理结构 | Physical structure |
| 线性结构/非线性结构 | Linear structure / Nonlinear structure |
| 线性表 | Linear list |
| 栈 | Stack |
| 队列 | Queue |
| 串 | String |
| 图 | Graph |
| 插入 | Insertion |
| 删除 | Deletion |
| 前趋 | Predecessor |
| 后继 | Successor |
| 直接前趋 | Immediate predecessor |
| 直接后继 | Immediate successor |
| 双端列表 | Double-ended queue |
| 循环队列 | Circular queue |
| 指针 | Pointer |
| 先进先出表 (队列) | First-in first-out list |
| 后进先出表 (队列) | Last-in first-out list |
| 栈底/栈顶 | Bottom / Top |
| 压入/弹出 | Push/ Pop |
| 队头/队尾 | Front/ Rear |
| 上溢/下溢 | Overflow/ Underflow |
| 数组 | Array |
| 矩阵 | Matrix |
| 多维数组 | Multi-dimensional array |
| 以行为主/以列为主的顺序分配 | Row major order / Column major order |
| 三角矩阵 | Triangular matrix |
| 对称矩阵 | Symmetric matrix |
| 稀疏矩阵 | Sparse matrix |
| 转置矩阵 | Transposed matrix |
| 链表 | Linked list |

| | |
|----------------|--|
| 线性链表 | Linear linked list |
| 单链表 | Single linked list |
| 多重链表 | Multilinked list |
| 循环链表 | Circular linked list |
| 双向链表 | Doubly linked list |
| 十字链表 | Orthogonal list |
| 广义表 | Generalized list |
| 指针域 | Pointer field |
| 头结点 | Head node |
| 头指针/尾指针 | Head pointer/ Tail pointer |
| 空白串 | Blank string |
| 空串（零串） | Null string |
| 子串 | Substring |
| 树 | Tree |
| 子树 | Subtree |
| 森林 | Forest |
| 根 | Root |
| 叶子 | Leaf |
| 深度 | Depth |
| 双亲/孩子/兄弟/祖先/子孙 | Parents/ Children/ Brother/ Ancestor/ Descendant |
| 二叉树 | Binary tree |
| 平衡二叉树 | Balanced binary tree |
| 满二叉树 | Full binary tree |
| 完全二叉树 | Complete binary tree |
| 遍历二叉树 | Traversing binary tree |
| 二叉排序树 | Binary sort tree |
| 二叉查找树 | Binary search tree |
| 线索二叉树 | Threaded binary tree |
| 霍夫曼树 | Huffman tree |
| 有序树/无序树 | Ordered tree / Unordered tree |
| 判定树 | Decision tree |
| 数字查找树 | Digital search tree |
| 树的遍历 | Traversal of tree |
| 先序遍历 | Preorder traversal |
| 中序遍历 | Inorder traversal |
| 后序遍历 | Postorder traversal |
| 子图 | Subgraph |
| 有向图/无向图 | Digraph (directed graph) /Undigraph (undirected graph) |
| 完全图 | Complete graph |
| 连通图 | Connected graph |
| 非连通图 | Unconnected graph |

| | |
|-------------|-----------------------------------|
| 强连通图 | Strongly connected graph |
| 弱连通图 | Weakly connected graph |
| 有向无环图 | Directed acyclic graph |
| 重连通图 | Biconnected graph |
| 二部图 | Bipartite graph |
| 边 | Edge |
| 顶点 | Vertex |
| 连接点 | Articulation point |
| 初始结点 | Initial node |
| 终端结点 | Terminal node |
| 相邻边 | Adjacent edge |
| 相邻顶点 | Adjacent vertex |
| 关联边 | Incident edge |
| 入度/出度 | In-degree/ Out-degree |
| 有序对/无序对 | Ordered pair/ Unordered pair |
| 简单路径 | Simple path |
| 简单回路 | Simple cycle |
| 连通分量 | Connected component |
| 邻接矩阵 | Adjacency matrix |
| 邻接表 | Adjacency list |
| 邻接多重表 | Adjacency multi-list |
| 遍历图 | Traversing graph |
| 生成树 | Spanning tree |
| 拓扑排序 | Topological sort |
| 偏序 | Partial order |
| AOV 网 | Activity On Vertex network |
| AOE 网 | Activity On Edge network |
| 关键路径 | Critical path |
| 线性查找 (顺序查找) | Linear search (Sequential search) |
| 二分查找 | Binary search |
| 分块查找 | Block search |
| 散列查找 | Hash search |
| 平均查找长度 | Average search length |
| 散列表 | Hash table |
| 散列函数 | Hash function |
| 直接定址法 | Immediately allocating method |
| 数字分析法 | Digital analysis method |
| 平方取中法 | Mid-square method |
| 随机数法 | Random number method |
| 内部排序 | Internal sort |
| 外部排序 | External sort |

| | |
|----------|------------------------------|
| 选择排序 | Selection sort |
| 基数排序 | Radix sort |
| 平衡归并排序 | Balance merging sort |
| 二路平衡归并排序 | Balance two-way merging sort |
| 多步归并排序 | Ployphase merging sort |
| 置换选择排序 | Replacement selection sort |
| 索引文件 | Indexed file |
| 索引顺序文件 | Indexed sequential file |
| 索引非顺序文件 | Indexed non-sequential file |
| 多重链表文件 | Multi-list file |
| 倒排文件 | Inverted file |

5.4.5 计算机网络相关术语

| | |
|-----------|--|
| 端口 | Port |
| 服务器 | Server |
| 客户端 | Client |
| 服务访问点 | SAP (Service Access Point) |
| 开放系统互联 | OSI (Open System Interconnection) |
| 物理层 | Physical layer |
| 数据链路层 | Data link layer |
| 网络层 | Network layer |
| 运输层 | Transport layer |
| 会话层 | Session layer |
| 表示层 | Presentation layer |
| 应用层 | Application layer |
| TCP/IP | TCP / IP protocol |
| 信道容量 | Channel capacity |
| 香农 | Shannon |
| 奈奎斯特 | Nyquist |
| 双绞线 | UTP (Unshielded Twisted Paired) |
| 同轴电缆 | Coaxial cable |
| 光纤 | Optical fiber |
| 不归零码 | NRZ (Non Return to Zero) |
| 曼彻斯特编码 | Manchester coding |
| 调制 | Modulation |
| 脉码调制 | PCM (Pulse Code Modulation) |
| 增量调制 | DM (Delta Modulation) |
| 同步传输/异步传输 | Synchronous transmission / ATM (Asynchronous transmission) |
| 循环冗余校验 | CRC (Cyclic Redundancy Check) |
| 流量控制 | Flow control |
| 滑动窗口 | Sliding window |

| | |
|----------------|---|
| 差错控制 | Error control |
| 帧结构 | Frame structure |
| 复用 | Reuse |
| 非对称数字用户线路 | ADSL (Asymmetric digital subscriber line) |
| 电路交换和分组交换 | Circuit switching and packet switching |
| 频分多路复用 | Frequency division multiplexing |
| 信令 | Signaling |
| 数据报 | Datagram |
| 虚电路 | Virtual circuit |
| 帧中继 | Frame relay |
| 信元 | Ceil |
| 拥塞 | Congestion |
| 反压 | Back pressure |
| 令牌桶 | Token bucket |
| 环形/总线型/树形和星形结构 | Ring/ bus/ tree and star structure |
| 局域网 | LAN (local area network) |
| 集线器 | Hub |
| 交换机 | Switch |
| 路由器 | Router |
| 网桥 | Network bridge |
| 以太网监听算法 | Ethernet listener algorithm |
| 子网掩码 | Subnet mask |
| 三次握手 | Three-way handshake |
| 地址解析协议 | APP (Address resolution protocol) |
| 瘦客户机 | Thin client |
| 网际控制报文协议 | ICMP (Internet Control Message Protocol) |
| 因特网群组管理协议 | IGMP (Internet Group Management Protocol) |
| 拒绝服务 | Denial of service |
| 边界网关 | Border gateway |
| 域名系统 | DNS (Domain Name System) |
| 数据链路控制 | DLC (Data Link Control) |
| 互联网电子邮件协议标准 | POP (Post Office Protocol) |
| 远程控制 | Remote control |
| 简单邮件传送协议 | SMTP (Simple Mail Transport Protocol) |

智力题攻略

第6章

而 IT 企业是一种智慧密集型企业，对人员的综合素质具有比较高的要求，优秀的程序员一般思维活跃、头脑冷静、逻辑推理能力强。而通过对专业知识的考查一般很难评判面试者的综合能力，而智力测试本身又是无法突击和准备的，对于评价一个人的思维能力具备一定的评价能力，所以智力题也成为很多 IT 企业面试笔试的热点。

6.1 推理类

逻辑推理就是把不同排列顺序的意识进行相关性的推导，找出正确的事件逻辑。很多时候编程的瓶颈不是编程语言，而是是否能够理清程序内部的业务逻辑，否则程序很难达到应用的标准。所以，对于系统开发而言，逻辑推理能力非常重要。

1. 猜数字

题目：一个教授逻辑学的教授有 3 个学生，他们个个都绝顶聪明。有一天，教授给他们出了一道题，教授在每个人的脑门上贴了一张纸条告诉他们：每个人的纸条上都写了一个正整数，且某两个数的和等于第三个。需要注意的是，每个人都可以看到其他两个人头上的数，却看不见自己的。教授问第一个学生：你能猜出自己的数吗？第一个学生回答：不能，教授接着问第二个学生，还是回答不能，接着问第三个学生，第三个学生的回答也是不能，等到教授再问第一个学生时，第一个学生再次回答不能，接着再问第二个学生，第二个学生的回答依然是不能，当问到第三个学生的时候，第三个学生回答：猜出来了，是 144。教授很满意地笑了。请问另外两个数是多少？

其他两个数，一个是 36，一个是 108。分析过程如下：经过第一轮，说明任何两个数都是不同的，因为如果有任何两个数相同的话，由于某两个数的和是第三个，所以第三个人都可以猜测到自己的数。

第二轮，前两个人没有猜出，说明任何一个数都不是其他数的两倍，因为如果有一个数是其他两个数的两倍的话，也与条件矛盾。

根据两轮分析，可以得出以下 3 个条件：

- (1) 每个数大于 0。
- (2) 任意两个数都不相等。
- (3) 任意一个数都不是其他数的两倍。

每个数字可能是另两个之和或之差，第三个人能猜出 144，必然根据前面 3 个条件排除了其中的一种可能。假设是两个数之差，即 $x-y=144$ 。这时 1 ($x, y>0$) 和 2 ($x \neq y$) 都满足，所以要否定 $x+y$ 必然要使 3 不满足，即 $x+y=2y$ ，解得 $x=y$ ，不成立（不然第一轮就可猜出），所以不是两数之差。因此是两数之和，即 $x+y=144$ 。同理，这时 1, 2 都满足，必然要使 3 不满足，即 $x-y=2y$ ，两方程联立，可得 $x=108, y=36$ 。

这两轮猜的顺序其实分别为这样：第一轮（一号，二号），第二轮（三号，一号，二号）。这样分大家在每轮结束时获得的信息是相同的（即前面的 3 个条件）。

那么就假设是 C，来看看 C 是怎么做出来的：C 看到的是 A 的 36 和 B 的 108，因为条件，两个数的和是第三个，那么自己要么是 72 要么是 144。

假设自己 (C) 是 72 的话，那么 B 在第二轮的时候就可以看出来，下面如果 C 是 72，B 的思路：这种情况下，B 看到的的就是 A 的 36 和 C 的 72，那么他就可以猜自己，是 36 或者是 108（猜到这个是因为 36 的话，36 加 36 等于 72，108 的话就是 36 和 108 的和）。

如果假设自己 (B) 头上是 36，那么 C 在第一轮的时候就可以看出来，下面如果 B 是 36，C 的思路：这种情况下，C 看到的的就是 A 的 36 和 B 的 36，那么他就可以猜自己，是 72 或者是 0（此处不再解释了）。

如果假设自己 (C) 头上是 0，那么 A 在第一轮的时候就可以看出来，假设 C 头上为 0 时，A 的考虑思路如下：A 看到的的就是 B 的 36 和 C 的 0，那么此时他就可以猜测自己头上的数据是 36（然后不停地逆推），现在 A 在第一轮没有报出自己的 36，反推，C（在 B 的想象中）就可以知道自己头上不是 0，如果其他和 B 的想法一样（指 B 头上是 36），那么 C 在第一轮就可以报出自己的 72。现在 C 在第一回合没报出自己的 36，B（在 C 的想象中）就可以知道自己头上不是 36，如果其他和 C 的想法一样（指 C 头上是 72），那么 B 在第二轮就可以报出自己的 108。现在 B 在第二回合没报出自己的 108，C 就可以知道自己头上不是 72，那么 C 头上的唯一可能就是 144 了。

2. 谁戴的黑帽子

题目：有一个牢房，有 3 个犯人被关在里面，因为玻璃很厚，所以 3 个人之间只能互相看见，不能听到对方说话的声音。有一天，国王想了一个办法，给他们每个人头上都戴一顶帽子，只让他们知道帽子的颜色不是白色的就是黑色的，不让他们知道自己所戴帽子的颜色，在这种情况下，国王宣布了两条规则：第一，谁能看到其他两个犯人戴的都是白帽子，就可以释放谁；第二，谁知道自己戴的是黑帽子，就释放谁。其实，国王给他们戴的都是黑帽子。他们因为被绑，看不见自己罢了，于是他们 3 个人互相盯着不说话，可是不久，心眼灵光的 A 用推理的方法，认定自己戴的是黑帽子，他是怎么推断的？

心眼灵光的 A 的推理如下：

首先来分析国王制定的两条规则：

(1) 谁能看到其他两个犯人戴的都是白帽子，就可以释放谁。

(2) 谁知道自己戴的是黑帽子，就释放谁。

在这 3 个人中间，如果存在两人戴白色帽子的情况，那么第三人立刻就可以释放，所以戴白色帽子的人数一定小于 2，即为 1 个或者 0 个。

下面分别对这两种情况进行分析。如果为 1，则戴黑色帽子的人看到的是一白一黑，自己会猜测如下：“我如果戴的是白色帽子，则有人看到两顶白色帽子已经被释放；如果我戴黑色帽子，则戴白色帽子的人看到两顶黑色帽子，而戴黑色帽子的人看到一黑一白两顶帽子。”两个戴黑色帽子的人都是这么想的，也就是说，假如真的只有一顶白色帽子，两顶黑色帽子，则一定有人可以推测出自己戴的是黑色帽子，但是两个戴黑色帽子的人都没有说话，说明该假设不成立。

所以只有一种可能，就是白色帽子数为 0。于是 A 断定，自己戴的就是黑色帽子。

3. 扑克牌花色

题目：S 先生、P 先生、Q 先生知道桌子的抽屉里有 16 张扑克牌：红桃 A、Q、4，黑桃 J、8、4、2、7、3，草花 K、Q、5、4、6，方块 A、5。约翰教授从这 16 张牌中挑出一张牌来，并把这张牌的点数告诉 P 先生，把这张牌的花色告诉 Q 先生。这时，约翰教授问 P 先生

和 Q 先生：你们能从已知的点数或花色中推知这张牌是什么牌吗？于是，S 先生听到如下的对话：P 先生：我不知道这张牌。Q 先生：我知道你不知道这张牌。P 先生：现在我知道这张牌了。Q 先生：我也知道了。听罢以上的对话，S 先生想了一想之后，就正确地推出这张牌是什么牌。请问：这张牌是什么牌？

答案是方块 5，推理过程如下：

首先，P 先生知道该牌的点数，但是 P 先生却说“我不知道这张牌”，说明他知道的点数在 4 个花色中一定不是只有一张。而对着 16 张扑克牌的花色进行分析，同一个点数却存在不同花色的有 A、Q、4、5，所以 P 先生知道的点数一定是 A、Q、4、5 之一。

其次，Q 先生知道该牌的花色，Q 先生说“我知道你不知道这张牌”，这句话说明该花色的每张牌的点数至少在其他 3 种花色之一都有重复，也就是说不可能是黑桃和草花（如 8 只有在黑桃中才有，如果 P 先生知道是 8 那他就知道该牌），只能是红桃和方块之一。

再次，P 先生通过上句 Q 先生的话，推理出花色一定是红桃和方块之一，于是 P 先生说“现在我知道这张牌了”，这说明该牌的点数一定在红桃与方块中是唯一的，于是排除 A，只能是 Q、4、5 之一。

最后，Q 先生说“我也知道了”，说明该牌除去花色 A 一定只剩一种可能了（否则 Q 先生不可能知道该牌具体是什么），所以只能是方块 5。

综合得出，该张牌为方块 5。

4. 有几条病狗

题目：村子里有 50 个人，每人有一条狗，在这 50 条狗中有病狗（这种病不传染），于是人们要找出病狗。每个人可以观察其他 49 条狗，以判断他们是否生病，（如果有病一定能看出来），只有自己的狗不能看，观察后得到的结果不得交流，也不能通知病狗的主人。主人一旦推算出自己家的狗是病狗就得枪毙自己的狗（发现后必须在一天内枪毙），而且每个人只有权利枪毙自己的狗，没有权利打死其他人的狗。第一天大家全看完了，但枪没有响，到了第三天传来一阵枪声，问村里共有几条病狗？

村里一共有 3 条病狗，推理过程如下：

首先假设只有一条病狗，狗的主人是 A，A 在第一天就会发现其他 49 条都是好狗，那么自己的狗一定是病狗。所以第一天 A 就会开枪杀了自己的狗。此时肯定会听到枪响，可是第一天未能听到枪响，所以此种假设不成立。

假设有两条病狗，狗的主人分别是 A 和 B，A 和 B 第一天都会猜测到底是有 1 条还是有两条病狗，因为他们都会看到 1 条病狗，但是不能确定自己的狗是否也生病了。第二天 A 发现 B 的狗没有被枪杀，B 发现 A 的狗也没有被枪杀，那么就能够确定一共有两条病狗，因为如果对方的狗没死，即表明对方也看到了一条病狗，所以对方不确定自己的是否是病狗，那么很明显对方看到的病狗是自己的，除了对方的以外，自己的也是。所以第 1 天后，即到第 2 天，A 和 B 会枪杀掉自己的狗。而其他的人也会在猜测到底是两条还是 3 条，但是他们的判断晚于 A 与 B。所以此种假设不成立。

假设有三条病狗，他们的主人分别是 A、B 和 C，每个病狗的主人只能看到两条病狗，他们会认为有两种可能：两条或 3 条病狗，根据第二条假设，如果是两条病狗的话，会在第二天听到枪响，结果第二天枪没响，所以两条病狗的假设不成立，只能是 3 条病狗。从 A 的角度来说，到第 3 天还发现 B 和 C 都没有杀自己的病狗，那么证明除了 B 和 C 以外还有一条病狗，那条病狗明显就是自己的。根据题目所给条件，第三天响起枪声，故 3 条病狗成立。所以一共有 3 条病狗。

假设有 4 条病狗，他们的主人分别是 A、B、C 和 D。A 第一天看到 3 条病狗，如果 A 认为自己的不是病狗，由第三条推理可知，第三天看时，那 3 条狗没死，所以病狗的数目肯定不是 3，而其他人没病狗，所以自己的狗必为病狗，故开枪；而 B、C、D 与 A 的想法一样，故也开枪。所以，如果为 4 条病狗，第三天看后 4 条狗必死。所以假设不成立。

不可能是 4 条以上的病狗，因为根据推理，如果有 4 条狗的话，那么肯定在第四天大家都可以判断出来。

所以一共有 3 条病狗。

5. 谁在说谎

题目：ABC 三个人都喜欢说谎，有时候也会说真话。一天，A 指责 B 说谎话，B 指责 C 说谎话，C 说 AB 两人都在说谎话，在他们三个人之中，至少有一个人说的是真话，请问谁说的是真话，谁在说谎？

B 说的是真话。A、C 在说谎。

可以采用假设的方法来进行推理。首先假设 A 说的是真话，那么 B 是说谎的，那么 C 无论说的是真话还是谎话，都不符合条件，所以 A 说真话的假设不成立，A 肯定说谎。此时，假设 B 说的是真话，那么 C 说谎，所以 A、C 在说谎，成立。假设 C 说的是真话，那么 A 与 B 都在说谎，可是 A 指责 B 说的是谎话，如果 A 在说谎，B 便是真话，此时与假设不相符合，所以 C 说谎。

如果 B 说的是真话，那么 A 指责 B 说谎，其实 A 就是在撒谎；而 C 说 A 与 B 都在说谎，可见 C 确实也说了谎，确认只有一个人说的是真话，那就是 B。所以最终可以确定 B 说的是真话，而 A 与 C 在说谎。

6. 经理女儿的年龄

题目：一个经理有 3 个女儿，3 个女儿的年龄加起来等于 13，3 个女儿的年龄相乘等于经理自己的年龄，有一个下属已经知道经理的年龄，但仍不能确定经理 3 个女儿的年龄，这时经理说只有一个女儿的头发是黑的，然后这个下属就知道了经理 3 个女儿的年龄。请问 3 个女儿的年龄分别是多少？为什么？

该经理有一对双胞胎女儿，她们的年龄分别是：2 岁、2 岁、9 岁；而经理的年龄是 36 岁。

因为 3 个女儿的年龄加起来等于 13，而且每个人的年龄都是整数，所以只要把 13 分成符合情理的 3 个数即可。3 数乘积等于经理的年龄，存在多种组合性，因为 3 个变量，只有两个方程，缺少条件，所以此时下属猜不出答案。

设 3 个女儿的年龄分别为：a，b，c；经理的年龄为 Y。则有如下两个式子：

式子 1： $a + b + c = 13$

式子 2： $a \times b \times c = Y$

按照常识应满足条件： $0 \leq a < Y < 100$ ， $0 \leq b < Y < 100$ ， $0 \leq c < Y < 100$ 。将所有的可能情况列出来，年龄组合情况见表 6-1。

表 6-1 年龄组合情况

| 年 龄 组 合 | 年 龄 之 和 | 年 龄 之 积 |
|----------|---------|---------|
| (11,1,1) | 13 | 11 |
| (10,2,1) | 13 | 20 |

(续)

| 年龄组合 | 年龄之和 | 年龄之积 |
|---------|------|------|
| (9,3,1) | 13 | 27 |
| (9,2,2) | 13 | 36 |
| (8,4,1) | 13 | 32 |
| (8,3,2) | 13 | 48 |
| (7,5,1) | 13 | 35 |
| (7,4,2) | 13 | 42 |
| (7,3,3) | 13 | 63 |
| (6,6,1) | 13 | 36 |
| (6,5,2) | 13 | 60 |
| (6,4,3) | 13 | 72 |

因为下属已知道经理的年龄，但仍不能确定经理 3 个女儿的年龄，说明经理是 36 岁，因为 $1 \times 6 \times 6 = 36$ ， $2 \times 2 \times 9 = 36$ ，而又由于只有一个女儿的头发是黑色的，说明只有一个女儿是比较大的，其他的都比较小，头发还没有长成黑色的，后两种情况均不符合实际情况，他的 3 个女儿的岁数为 9 岁，2 岁，2 岁，符合题目条件。

所以，此经理有一对双胞胎女儿，她们的年龄分别是：2 岁、2 岁、9 岁；而经理的年龄是 36 岁。

7. 该走哪条路

题目：一个岔路口分别通往诚实国和说谎国，来了两个人，已知一个是诚实国的，另一个是说谎国的。诚实国永远说实话，说谎国永远说谎话。现在需要去说谎国，但是不知道该走哪条路，需要对问这两个人来获取答案，请问如何提问才能找出去说谎国的路。

由于两个人，一个人说假话，一个人说真话，所以当随便对两个人提问：“你的国家往哪里走”时，诚实国的人肯定会指向去往诚实国的路，而说谎国的人因为会说谎，所以不会指向去自己的说谎国的路，肯定也会指向去往诚实国的路，所以可以很快地确定另一条路就是去往说谎国的路了。

8. 谁是金牌

题目：数学竞赛后，小明、小华和小强各获得一枚奖牌，其中一人得金牌，一人得银牌，一人得铜牌。老师猜测：“小明得金牌，小华不得金牌，小强不得铜牌。”结果老师只猜对了一个，那么谁得金牌，谁得银牌，谁得铜牌？

小华得金牌，小强得银牌，小明得铜牌。

可以采用假设的方式来进行矛盾推理，如果小明得金牌，那么小华一定“不得金牌”，这与“老师只猜对了一个”相矛盾，所以假设不成立。假设小华得金牌，那么“小明得金牌”与“小华不得金牌”这两句都是错的，“小强不得铜牌”应是正确的，所以小强得银牌，小明得铜牌。假设“小强得金牌”，那么“小明得金牌”是错误的，“小华不得金牌”与“小强不得铜牌”都是对的，与“老师只猜对一个”矛盾，所以此种假设不成立。

9. 谁是木匠

有 4 个朋友住在一个小城镇里。他们的名字是库克、米勒、史密斯、卡特。他们一个是警察、一个是木匠，一个是农民，一个是医生。一天，库克的儿子摔断了腿。库克带他去找医生。医生有个妹妹是史密斯的妻子。农民没有结过婚，他养着许多母鸡。米勒经常去农民家里买鸡蛋。警察每天都能见到史密斯，因为他们是邻居。请问，他们 4 人中，谁是警察？谁是

木匠？谁是农民？谁是医生？

史密斯是木匠，库克是警察，米勒是医生，卡特是农民。

在没有任何条件的情况下，每种对应关系都有可能，用下面的方式表示他们的所有可能情况：

库克：警察，木匠，农民，医生

米勒：警察，木匠，农民，医生

史密斯：警察，木匠，农民，医生

卡特：警察，木匠，农民，医生

根据题目条件，首先库克去找医生，可以知道，库克肯定不是医生。然后，医生有个妹妹是史密斯的妻子，可以表明史密斯也不是医生。接着，农民没有结过婚，说明库克和史密斯都不是农民，因为库克有儿子，史密斯有妻子，可以推测他们都是结过婚的人。然后，米勒经常去农民家里买鸡蛋，说明米勒不是农民。最后，警察每天都能见到史密斯，可以说明史密斯不是警察。通过排除法，可能性范围逐步缩小，变成了如下情况：

库克：警察，木匠

米勒：警察，木匠，医生

史密斯：木匠

卡特：警察，木匠，农民，医生

通过排除法，最后可以把每个人的角色找出来。史密斯只有一种可能是木匠，当史密斯是木匠的时候，其他人为木匠的可能性被排除，进而推出库克是警察，此时，其他人为木匠、警察的可能性也被排除，接着可以推测出米勒是医生，卡特是农民。

所以最后的结果是史密斯是木匠，库克是警察，米勒是医生，卡特是农民。

10. 取药问题

题目：A、B 两人分别在两座岛上。B 生病了，A 有 B 所需要的药。C 有一艘小船和一个可以上锁的箱子。C 愿意在 A 和 B 之间运东西，但东西只能放在箱子里。只要箱子没被上锁，C 就会偷走箱子里的东西，不管箱子里有什么。如果 A 和 B 各自有一把锁和只能开自己那把锁的钥匙，A 应该如何把东西安全递交给 B？

A 把药放进箱子，用自己的锁把箱子锁上。B 拿到箱子后，再在箱子上加一把自己的锁。箱子运回 A 后，A 取下自己的锁。箱子再运到 B 手中时，B 取下自己的锁，获得药物。

11. 选择游戏

题目：如果从下面两种游戏中选择一种，你选择哪一种？

第一种，写下一句话。如果这句话为真，你将获得 10 美元；如果这句话为假，你获得的金钱将少于 10 美元或多于 10 美元（但不能恰好为 10 美元）。第二种，写下一句话。不管这句话的真假，你都会得到多于 10 美元的钱。

选择第一种游戏，并写下“我既不会得到 10 美元，也不会得到 10000000 美元”。此时，如果这句话为真，得到 10 美元与这句话的内容矛盾，不成立。如果这句话为假，他将获得少于 10 美元或多于 10 美元的钱数，而这句话内容为假时，则此人可能得到 10 美元，也可能得到 10000000 美元，最终此人得到的金钱数只能是 10000000 美元。第二种方式，只是提及了多于 10 美元。

6.2 博弈类

对于博弈类的问题，一般而言（有时不是，但思路类似），题目要求谁先下手谁赢。这就

要求第一个行动的人行动之后，在剩下的局面中，对方任何一个举动，自己都有相应的应对策略，且按照该策略进行下去，自己一定会赢。

按照上面的分析，胜者需要确定两个关键因素：（1）确定谁先行动，如果是自己，第一步该怎么走？（2）针对对方所有可能的行动，制定相应的应对策略。

1. 硬币问题

题目：考虑一个双人游戏，游戏在一个圆桌上进行，每个玩游戏的人都有足够多的硬币，他们需要在桌子上轮流放置硬币，每次必需放一枚硬币，而且只能放置一枚硬币，同时要求硬币完全置于桌面内（注意，不能有一部分悬在桌子外面），并且不能与原来放过的硬币重叠。谁没有地方放置新的硬币，谁就输了。游戏的先行者还是后行者有必胜策略？如果有，这种策略是什么？

游戏的先行者具有必胜策略。策略如下：首先先行者在桌子中心放置一枚硬币，以后的硬币总是放在与后行者刚才放的地方相对称的位置。这样，只要后行者能放，先行者一定也有地方放，直到后行者没地方放了，游戏结束，先行者胜出。

还有一类硬币问题：16 个硬币，A 和 B 轮流拿走一些，每次拿走的个数只能是 1, 2, 4 中的一个数。谁最后拿硬币谁输。问：A 或 B 有无策略保证自己赢？假设都很聪明。

该题的思路其实也很明了，由于本题要求每人拿球的数目必须满足为 1, 2, 4 中的一个数，考虑边界情况，让对方先拿，而自己保证每一轮拿掉的硬币总数是 3 或者 6，即对方拿 1 个则自己拿 2 个，对方拿 2 个则自己拿 1 个或 4 个，对方拿 4 个则自己拿 2 个，如此这般，由于 $16\%3=1$ ，则经过若干轮后，最后剩余 1 个，被对方拿走。

2. 乒乓球问题

题目：假设排列着 100 个乒乓球，由两个人轮流拿球装入口袋，能拿到第 100 个乒乓球的人为胜利者。条件是每次拿球者至少要拿 1 个，但最多不能超过 5 个。问：如果你是最先拿球的人，你该拿几个？以后怎么拿就能保证你能得到第 100 个乒乓球？

谁先拿球谁赢，如果想获胜，必须确保：自己第一次拿走一些乒乓球之后，在剩下的局面中，对方拿走一定数目的球，自己均有一定策略拿走与之对应数目的球。

本题要求每人每次拿走球的数目满足区间[1,5]，只考虑边界情况（ $1+5=6$ ），只要对方拿走 x 个，自己则拿走 $6-x$ 个，这样每一轮会拿走 6 个球，共有 $100/6=16$ 轮，最终剩下 $100\%6=4$ ，因而自己开始拿走 4 个球即可保证自己胜。

3. 海盗分金问题

题目：有 5 个海盗，按照等级从 5 到 1 排列。最大的海盗有权提议他们如何分享 100 枚金币。但其他人要对此表决，如果多数人（所有人中的多数）反对，那他就会被杀死。他应该提出怎样的方案，既让自己拿到尽可能多的金币又不会被杀死？

分配方案是 98, 0, 1, 0, 1。

5 级海盗会不会被杀死，取决于 5 级海盗死后其他海盗是否会获得更多的利益。如果可以获得更多的利益，则其他海盗肯定会反对；如果会获得更少的利益，则其他海盗肯定会支持，如果利益没有变化，则反对或支持都可以。

如果 5 级海盗死了，则由 4 级海盗分配，4 级海盗面临同样的问题，需要看自己死后的利益分配变化。然后是 3 级海盗，2 级海盗。

2 级海盗无论提出什么方案，都不会有多数人反对（自己支持，另一个人反对不能构成多数反对）。所以 2 级海盗肯定会提出 100, 0 的分配方案，自己独享所有金币。猜到 2 级海盗的分配方案后，3 级海盗会提出 99, 0, 1 的分配方案。这样 1 级海盗因获得了比 2 级海盗方案

中更多的金币，所以会支持3级海盗的方案。猜到3级海盗的分配方案后，4级海盗会提出99, 0, 1, 0的分配方案。这样2级海盗获得了比3级海盗方案中更多的金币，所以会支持4级海盗的方案。猜到4级海盗的分配方案后，5级海盗会提出98, 0, 1, 0, 1的分配方案。这样1级海盗和3级海盗获得了比4级海盗方案中更多的金币，所以会支持5级海盗的方案。

6.3 计算类

数学是程序的灵魂，是计算机的基础。程序员需要一定的数学修养，不仅仅是编码本身的需要，通过学习数学可以锻炼自己的思维能力，进而解决现实中的问题。

“我不要当码农，请叫我工程师”，说出了无数程序员的心声。其实码农与工程师是有区别的，普通的“码农”可能只需要会编码即可，但是要想成为一名有潜力、有发展前途的能够称为工程师的程序员，就更加需要对数学能力的培养。

1. 小鸟飞行距离

题目：有一辆火车以15km/h的速度离开洛杉矶直奔纽约，另一辆火车以20km/h的速度从纽约开往洛杉矶。如果有一只鸟，以30km/h的速度和两辆火车同时启动，从洛杉矶出发，碰到另一辆车后返回，依次在两辆火车间来回飞行，直到两辆火车相遇，请问，这只小鸟飞行了多长距离？

设纽约到洛杉矶的距离为S1，因为鸟是不停地在飞行，而车相遇的时间就是鸟飞行的时间，设小鸟飞行的时间为t，小鸟飞行的距离为S2，则有如下关系式。

$$t = S1 / (15 + 20)$$

$$S2 = t \times 30 = S1 \times 6/7$$

即鸟飞的距离是纽约到洛杉矶路程的6/7。

2. 分金条

题目：工人工作7天，给工人的回报是一根金条，金条平分成相连的7段，必须在每天结束时给他们一段金条，如果只允许两次把金条弄断，如何给工人付费？

根据题目要求，两次弄断就应该只能将金条分成3份，把金条分成1/7、2/7和4/73份可以满足要求。第1天可以给工人1/7；第2天给工人2/7，让工人找回第一天给的1/7；第3天给工人1/7，加上原先的2/7，此时工人手里的金条为3/7；第4天给工人那块4/7的金条，让工人找回那两块1/7和2/7的金条；第5天，给工人1/7；第6天和第2天一样；第7天给工人找回的那个1/7。

3. 时针分针重合

假设时钟到了12点。注意时针和分针重叠在一起。在一天之中，时针和分针共重叠多少次？

11次。

记00:00:00和11:59:59两个边界值，把[00:00:00, 11:59:59]作为一个求解区间。假设某一时刻时针和00:00:00时针的顺时针方向夹角为 x° ，则此时分针和00:00:00时针的顺时针方向夹角为 $12x - n \times 360^\circ$ （ n 为使 $12x - n \times 360$ 大于0且小于等于360的最小自然数）。那么根据条件就有方程： $x = 12x - n \times 360$ ，

解此方程， x 的值满足 $\{360/11, 720/11, 1080/11, 1440/11, 1800/11, 2160/11, 2520/11, 2880/11, 3240/11, 3600/11, 3960/11\}$ ，即约 $\{32.7, 65.5, 98.2, 130.9, 163.6, 196.4, 229.1, 261.8, 294.5, 327.3,$

360}, 将该夹角度数值转换为对应的秒数满足公式: 时间秒数 $t=x/360 \times 12 \times 60 \times 60$, 带入 x 的值, t 的值变为 {3927.3, 7854.5, 11781.8, 15709.1, 19636.4, 23563.6, 27490.9, 31418.2, 35345.5, 39272.7, 43200.0}, 即在 12 个小时内, 分别在以上 11 个时间点时针与分针重合。

4. 烧绳记时

题目: 烧一根不均匀的绳要用一个小时, 如何用它来判断一个小时 15 分钟?

一共需要 3 根绳子。

由于绳子是双向的, 首先取两个相同的绳子, 其中一根绳子从两头点燃, 另一根绳子只点燃一头。当第一根绳子燃烧完时, 由于是从两头开始燃烧的, 所以耗时为半小时, 而此时第二根绳子还未烧尽, 此时将第二根绳子的另一头点燃, 并开始计时。则从计时开始到第二根绳子燃烧完一共用时 15 分钟。再取第三根绳子从两头点燃, 直至这根绳子燃烧完, 计时结束。则从计从开始到计时结束, 用时 30 分钟+15 分钟+30 分钟, 合计 75 分钟, 即一个小时 15 分钟。

5. 猴子分桃

题目: 5 只猴子分一堆桃子, 怎么也分不成 5 等份, 只好先回去睡觉, 准备第二天再分。半夜里有只猴子起来把一个桃子扔掉, 刚好可以分成 5 份, 它把自己的那份先收起来, 第二只猴子起来也是把一个桃子扔掉, 刚好也可以分成 5 份, 它也把自己的那份收起来; 后来第三、第四、第五只猴子也轮流起来依次这样做, 最后桃子刚好可以分完, 请问这堆桃子最少有多少个?

这堆桃子最少有 3121 个。

假设最开始一共有 x 个桃子, 此时把 x 变换为 $(x+4)-4$, 当第一个猴子起来扔掉 1 个, 还有桃子 $(x+4)-4-1=(x+4)-5$ 个桃子, 这时恰好可分成 5 份, 每份的桃子数为 $[(x+4)-5]/5=(x+4)/5-1$ 个, 其中桃子数不可能为小数, 所以 $(x+4)/5$ 必须为整数, 所以 $(x+4)$ 是 5 的倍数, 当第一个猴子藏掉一份后, 剩下的桃子为: $(4/5) \times [(x+4)-5] = (4/5) \times (x+4)-4$ 。同样, 第二只猴子来了, 一扔一藏之后, 剩下的桃子数为 $(4/5) \times [(4/5) \times (x+4)-5]$, 由于 $(4/5) \times (4/5) \times (x+4)$ 是整数, 故 $(x+4)$ 应是 $5 \times 5 = 25$ 的倍数, 如此一来 5 只猴子一扔一藏, 恰好剩下 $(4/5) \times (4/5) \times (4/5) \times (4/5) \times (4/5) \times (x+4)-5$ 个桃子, 故 $(x+4)$ 必须是 $5 \times 5 \times 5 \times 5 \times 5$ 的倍数, 即 $x+4=5^5$, 所以: $x=3125-4=3121$, 即开始最少有 3121 个桃子。

6. 10 匹马取前 5

题目: 一共有 25 匹马, 有一个赛场, 赛场有 5 个赛道, 就是说最多同时可以有 5 匹马一起比赛。假设每匹马都跑得很稳定, 不用任何其他工具, 只通过马与马之间的比赛, 试问最少得比赛多少场才能知道跑得最快的 5 匹马?

最少 7 场比赛, 最多 8 场比赛就可确定跑得最快的 5 匹马。

首先将 25 匹马分成 5 组, 进行 5 场比赛。第六场比赛可以考虑都取各个小组的第一名 (或第二名)。假设都取各小组的第一名, 根据这场比赛的排名, 将原来的小组分别编号为 a、b、c、d、e, 并将原来的 25 匹马分别编号为:

| | | | | |
|----|----|----|----|----|
| a1 | b1 | c1 | d1 | e1 |
| a2 | b2 | c2 | d2 | e2 |
| a3 | b3 | c3 | d3 | e3 |
| a4 | b4 | c4 | d4 | e4 |
| a5 | b5 | c5 | d5 | e5 |

用 X_i 表示, 其中 X 表示组的编号, i 为在该组的排名, 则有:

$a_1 > b_1 > c_1 > d_1 > e_1$

$a_1 > a_2 > a_3 > a_4 > a_5$

$b_1 > b_2 > b_3 > b_4 > b_5$

...

$e_1 > e_2 > e_3 > e_4 > e_5$

注意到: 跑得比 a_3 、 b_2 、 c_1 这三匹马都快的只可能是 a_1 、 a_2 、 b_1 , 因而 a_3 、 b_2 、 c_1 三匹马中跑得最快的必然是前四之一。因此, 第七场比赛, 这三匹马必然参加, 剩下两个名额待定。先考虑这三匹马的排名:

下面用 $[]$ 集合表示已确定是前五的马, 用 $\{\}$ 集合表示剩下的马中所有可能是前五的马。

(1) $a_3 b_2 c_1$ 或 $a_3 c_1 b_2$: 则 $[a_1, a_2, a_3] + \{a_4, a_5, b_1, b_2, c_1\}$

(2) $b_2 a_3 c_1$: $[a_1, b_1, b_2] + \{a_2, a_3, b_3, b_4\}$

(3) $b_2 c_1 a_3$: $[a_1, b_1, b_2] + \{a_2, b_3, b_4, c_1, c_2, d_1\}$

(4) $c_1 a_3 b_2$: $[a_1, b_1, c_1] + \{a_2, c_2, c_3, d_1, d_2, e_1\}$

为了能在第八场确定前五, 必须将上面的 $\{a_2, b_3, b_4, c_1, c_2, d_1\}$ 和 $\{a_2, c_2, c_3, d_1, d_2, e_1\}$ 的候选马匹数减少到 5 匹, 因而剩下的两个名额必须是这两个集合的重复元素, 即是 $\{a_2, c_2, d_1\}$ 中的两个。由于 a_2 跑得比 a_3 快, 若选择 a_2 的话, 不能利用前面的分析, 因而剩下两匹马选择 c_2 和 d_1 。

第七场比赛: a_3 、 b_2 、 c_1 、 c_2 、 d_1 的前两名是:

(1) a_3 : $[a_1, a_2, a_3] + \{a_4, a_5, b_1, b_2, c_1\}$ 的前两名 (由第八场比赛决定)

(2) $b_2 a_3$: $[a_1, b_1, b_2] + \{a_2, a_3, b_3, b_4\}$ 的前两名

(3) $b_2 c_1$: $[a_1, b_1, b_2] + \{a_2, b_3, b_4, c_1, \max(c_2, d_1)\}$ 的前两名

(4) $c_1 a_3$: $[a_1, a_2, a_3, b_1, c_1]$ (第七场就可确定前五)

(5) $c_1 b_2$: $[a_1, b_1, c_1] + \{a_2, b_2, b_3, c_2, d_1\}$ 的前两名

(6) $c_1 c_2$: $[a_1, b_1, c_1, c_2] + \{a_2, b_2, c_3, d_1\}$ 的第一名

(7) $c_1 d_1$: $[a_1, b_1, c_1, d_1] + \{a_2, b_2, c_2, d_2, e_1\}$ 的第一名

因而, 最少 7 场比赛, 最多 8 场比赛就可确定跑得最快的 5 匹马。

6.4 作图类

空间想象能力是一名优秀的程序员应该具备的基本能力, 而通过考查求职者作图类的问题, 一般也能起到考查求职者该方面能力的作用。

1. 种树问题

题目: 怎样种植 4 棵树木, 使其中任意两棵树的距离相等。

本题如果只是考虑一个平面的话, 是无法做到的, 所以需要发散思维。对于图 6-1 中的正四面体而言, 每个顶点之间的距离相等, 所以本题的答案是可以考虑在山坡、丘陵等可以组成正四面体结构的地带来种植树木。

2. 九点十线

题目: 九个点, 十条线, 要求每条线经过三个点, 能画出来吗?

可以, 首先把点摆成 3×3 的方阵, 把中间一行的两侧的点分别向里移至两点间距的 $1/2$ 处, 此时的九点, 可画出十条直线, 每条直线上三点。具体图形如图 6-2 所示。

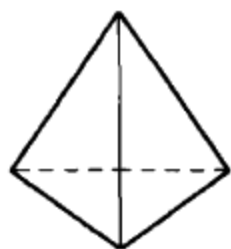


图 6-1 正四面体

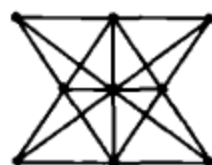


图 6-2 九点十线

需要注意的是，上述图形经过逆时针或顺时针旋转可以得到另外的表现形式，也能满足题目要求，也是正确的结果。

6.5 倒水类

1. 量水

题目：有无穷多水，一个 3L 和一个 5L 的桶，如何能够准确地称量出 4L 的水？

首先用水把 5L 的桶装满，然后把 5L 水倒入 3L 的桶中，直到把 3L 的桶装满为止，此时 5L 桶里只剩余 2L 水，把 3L 桶中水清空，再把 5L 桶中剩余的水倒入 3L 桶中，此时 3L 桶内有 2L 水，而 5L 桶内没有水，把 5L 桶装满水，再把 5L 桶中的水倒入 3L 桶，直到 3L 桶满，此时 5L 桶内剩余的水只有 4L，即为所求结果。

2. 倒酒

题目：有 3 个酒杯，其中两个大酒杯每个可以装 8 两酒，一个可以装 3 两酒。现在两个大酒杯都装满了酒，只用这 3 个杯子怎么把酒平均地分给 4 个人喝？

为了便于说明，用如下 3 个代号表示：A-X、B-X、C-X。其中，A 和 B 表示两个 8 两的酒杯，C 表示 3 两的酒杯，X 表示酒的数量。→表示一个操作过程。甲、乙、丙、丁分别表示 4 个人。

第一步，从 A 中倒 3 两酒到 C 杯中，甲喝 C 杯中 3 两酒：A-8、B-8、C-0 → A-5、B-8、C-3 → A-5、B-8、C-0。

第二步，从 A 中倒 3 两酒到 C 杯中，乙喝 A 杯中的 2 两酒：A-5、B-8、C-0 → A-2、B-8、C-3 → A-0、B-8、C-3。

第三步，将 C 杯中的 3 两酒倒入 A 中，将 B 中倒 3 两酒到 C 中，再将 C 杯中的 3 两酒倒往 A 杯中，再将 B 杯倒 3 两酒到 C 杯中，再将 C 杯到 2 两酒到 A 杯中，此时甲喝掉 C 杯中剩余的 1 两酒，两次一共喝了 4 两酒：A-0、B-8、C-3 → A-3、B-8、C-0 → A-3、B-5、C-3 → A-6、B-5、C-0 → A-6、B-2、C-3 → A-8、B-2、C-1 → A-8、B-2、C-0。

第四步，将 B 杯中的 2 两酒倒入 C 杯中，然后将 A 杯中到 1 两酒到 C 杯中，再将 C 杯中的 3 两酒倒入 B 杯中，再从 A 杯中倒 3 两酒到 C 杯中，再将 C 杯的 3 两酒倒入 B 杯中，再将 A 杯中的酒倒 3 两到 C 杯中，此时 A 杯剩余 1 两酒，B 杯剩余 6 两酒，C 杯剩余 3 两酒，丙喝掉 A 杯中的 1 两酒：A-8、B-2、C-0 → A-8、B-0、C-2 → A-7、B-0、C-3 → A-7、B-3、C-0 → A-4、B-3、C-3 → A-4、B-6、C-0 → A-1、B-6、C-3 → A-0、B-6、C-3。

第五步，将 C 杯中的 2 两酒倒入 B 杯中，此时 C 杯剩余 1 两酒，丁喝掉 C 杯中的 1 两酒：A-0、B-6、C-3 → A-0、B-8、C-1 → A-0、B-8、C-0。

第六步，从 B 杯往 C 杯中倒 3 两酒，丙喝掉 C 杯中的 3 两酒，两次一共喝了 4 两酒：A-0、B-8、C-0 → A-0、B-5、C-3 → A-0、B-5、C-0。

第七步，从 B 杯往 C 杯中倒 3 两酒，乙喝掉 B 杯中 2 两酒，两次一共喝了 4 两酒，丁喝 C 中 3 两酒，两次一共喝了 4 两酒：A-0、B-5、C-0→A-0、B-2、C-3→A-0、B-0、C-0。

此时分酒完成，最终每人喝掉 4 两酒。

6.6 称重类

1. 天平称重

题目：有 7 克、2 克砝码各一个，天平一只，如何只用这些物品 3 次将 140 克的盐分成 50、90 克各一份？

可以采用如下步骤将 140 克的盐分成 50 克、90 克各一份，而且只需要 3 次。

(1) 首先把 140 克盐等分两份放在天平两边，此时每边盐的重量均为 70 克。

(2) 然后把其中一份 70 克等分成两份，放在天平两边，此时每边盐的重量均为 35 克。

(3) 再次把 35 克盐放在天平一端，同时把 7 克砝码也放在该端，2 克砝码放在天平另一端，然后把天平上的 35 克盐往放 2 克砝码那边分，直到天平平衡，此时左右两边盐与砝码的重量均为 $(35 \text{ 克} + 7 \text{ 克} + 2 \text{ 克}) / 2 = 22 \text{ 克}$ 。

(4) 经过步骤 (3)，天平放 2 克砝码端的盐重 20 克，将 20 克盐与步骤 1 的 70 克盐混合即为 90 克，其他盐混合即为 50 克。

2. 被污染的罐子

题目：有 4 个装药丸的罐子，每个药丸都有一定的重量，被污染的药丸是没被污染的重量 +1，只称量一次，如何判断哪个罐子的药被污染了？

给 4 个不同的药丸罐子分别编号为 1、2、3、4，设药丸重 1 个单位，从 1 号罐中取出 1 颗，2 号罐取出 2 颗，3 号罐取出 4 颗，4 号罐取出 8 颗，如果都没被污染应该是 15。

表 6-2 中左侧表示被污染的罐号，右侧表示称得实际重量。

根据称得的重量，对应上表即知哪些罐被污染了。

引申：有 5 瓶药，每个药丸重 10 克，只有一瓶受到污染的药丸重量发生了变化，每个药丸重 9 克。给一个天平，怎样一次就能测出哪一瓶是受到污染的药呢？

分别给 5 个瓶子编号 1、2、3、4、5。从 1 号瓶中取 1 个药丸，2 号瓶中取 2 个药丸，3 号瓶中取 3 个药丸，4 号瓶中取 4 个药丸，5 号瓶中取 5 个药丸。把它们全部放在天平上称一下重量。现在用 $1 \times 10 + 2 \times 10 + 3 \times 10 + 4 \times 10 + 5 \times 10$ 的结果减去测出的重量。结果就是装着被污染的药丸的瓶子号码。

3. 找最重的球

题目：假设有 8 个球，外表一模一样，但是其中一个略微重一些，要找出这个球的唯一办

表 6-2 污染情况表

| 被污染的罐号 | 实际重量 |
|---------|------|
| 1 | 16 |
| 2 | 17 |
| 3 | 19 |
| 4 | 23 |
| 1、2 | 18 |
| 1、3 | 20 |
| 1、4 | 24 |
| 2、3 | 21 |
| 2、4 | 25 |
| 3、4 | 27 |
| 1、2、3 | 22 |
| 1、2、4 | 26 |
| 1、3、4 | 28 |
| 2、3、4 | 29 |
| 1、2、3、4 | 30 |

法是将两个球放在天平上，最少需要称量多少次能找出这个较重的球？

最少需要两次可以找出比较重的球，原因如下。

首先将 8 个球分成三组，第一组 3 个球，第二组 3 个球，第三组 2 个球，首先把 3 个球的两组分别放到天平两边，这时可能存在两种情况：

(1) 两边一样重。此时可以表明，稍重的球在另外只有两个球的组中，再通过天平称量一次，即可找出该球。

(2) 两边不一样重。此时把稍重的组随意挑选个 2 个球分别放在天平两边即可知道稍重的球。如果两个球一样重，则第三个球肯定是稍重的球，如果两个球中有一个球稍重，则该球就是要找的球。

4. 分药

题目：某种药方要求非常严格，每天需要同时服用 A、B 两种药片各一颗，不能多也不能少。这种药非常贵，不希望有任何一点的浪费。一天，打开装药片 A 的药瓶，倒出一粒药片放在手心；然后打开另一个药瓶，但不小心倒出了两粒药片。现在，手心上有一颗药片 A，两颗药片 B，并且无法区别哪个是 A，哪个是 B。如何才能严格遵循药方服用药片，并且不能有任何的浪费？

把手上的三片药各自切成两半，分成两堆摆放，此时每堆中有两个半粒的 B 药（合计为一颗 B 药的量），1 个半粒的 A 药，但是无法区分谁是 A，谁是 B。再取出一粒药片 A，也把它切成两半，然后在每一堆里加上半片的 A，此时，每一堆药片恰好包含两个半片的 A 和两个半片的 B，合计正好为 1 颗药片 A 与一颗药片 B 的量，所以一天服用其中的一堆即可。

6.7 最优化类

1. 猴子搬香蕉

题目：一只小猴子旁边上有 100 根香蕉，它要走过 50m 才能到家，每次它最多搬 50 根香蕉，每走 1m 就要吃掉一根，请问它最多能把多少根香蕉搬到家里（提示：猴子可以把香蕉放下往返地走，但是必须保证它每走一米都能有香蕉吃）。

16 根。

由于猴子一次最多只能搬 50 根香蕉，则它需要在中途返回一趟去取剩下的香蕉。设猴子在中途 x m 处返回，则最优的情况是：在 x m 处需贮存至少 50 根香蕉，然后义无反顾地回家（不再返回）。显然， $(50-2x) + (50-x) \geq 50$ ，求得 $x \leq 16.6$ ，则 x 取 16，此时猴子可以带回家 16 根香蕉。

具体过程如下：猴子先搬 50 根，当走 17m 时，吃掉 17 根香蕉，再回来搬 50 根走到 17 米处，分两次吃掉 34 根香蕉，回来后再搬剩下的 49 根走完 33m，路上吃掉 33 根香蕉，最后得到 16 根。

2. 买汽水

题目：1 元钱可以买一瓶汽水，喝完后两个空瓶换一瓶汽水，有 20 元钱，最多可以喝到几瓶汽水？

因为题目没有规定是否可以向商贩借瓶子，所以可以分为以下两种情况进行讨论。

(1) 如果不可以向卖汽水的商贩借瓶子，第一次可以用 20 元钱买 20 瓶汽水，喝完后有 20 个空瓶子；第二次用这 20 个空瓶子换 10 瓶汽水，喝完后剩余 10 个空瓶子；第三次用第二次剩余的这 10 个空瓶子换 5 瓶汽水，喝完后剩余 5 个空瓶子；第四次可以用第三次剩余的 5

个空瓶子换 2 瓶汽水，喝完后剩余 3 个空瓶子；第五次将第四次的 3 个空瓶子换 1 瓶汽水，多余一个空瓶子，将换的那瓶汽水喝掉，此时剩余 2 个空瓶子；第六次将第五次的 2 个空瓶子换成一瓶汽水，喝完后剩余 1 个空瓶子。所以一共是 $20+10+5+2+1+1=39$ 瓶。

(2) 如果可以借一个瓶子的话，在上述推导中，最后一次剩余的空瓶子再借一个空瓶子，正好可以换一瓶汽水，喝完后将瓶子换回，所以最后是 $20+10+5+2+1+1+1=40$ 瓶。

3. 取钻石

题目：一楼到 10 楼的每层电梯门口都放着一颗钻石，钻石大小不一。你乘坐电梯从一楼到 10 楼，每层楼的电梯门都会打开一次，只能拿一次钻石，问怎样才能拿到最大的一颗？

本题是一道开放式题目，没有标准答案，它考查的是求职者的开放性思维和逻辑推理能力。

以下是一些比较有代表性的回答方式：

(1) 选择前五层楼都不拿，观察各层钻石的大小，做到心中有数。后面五个楼层再选择，选择大小接近前五层楼出现过最大钻石大小的钻石。

(2) 对前三层进行比较，对于前三个中的最大有了一个概念，然后中间三个楼层作参考，以此确定最大的一颗的平均水平，在最后剩下的 4 颗中选择一颗最大的。

需要注意的是，本题无论采用何种方式都无法保证 100% 地拿到最大的一颗钻石。

6.8 IT 思想类

1. 台阶问题

题目：一个人上台阶可以一次上一个或两个，问这个人上 n 层的台阶，一共有多少种走法。

本题可以采用递归的方法来设计模型，先从数字的规律入手：假设共有 i 阶台阶，走完所有的台阶有 n 种走法，则存在的组合情况见表 6-3。

表 6-3 组合情况

| i | n | 组 合 情 况 |
|-------|--------------------------|---|
| 1 | 1 | {1} |
| 2 | 2 | {1, 1} {2} |
| 3 | 3 | {1, 1, 1} {1, 2} {2, 1} |
| 4 | 5 | {1, 1, 1, 1} {1, 1, 2} {1, 2, 1} {2, 1, 1} {2, 2} |
| ... | ... | ... |
| $n-2$ | $F(n-2)$ | |
| $n-1$ | $F(n-1)$ | |
| n | $F(n) = F(n-1) + F(n-2)$ | |

根据递推可以知道， $F(n) = F(n-1) + F(n-2)$ 。此式很熟悉，为常见的 Fibonacci 数列，此处不再赘述其求解算法。

2. 电线线头问题

题目：在一幢 100 层大楼下，有 21 根电线线头分别标有数字 1~21。这些电线一直延伸到大楼楼顶，楼顶的线头处标有字母 A~U，一共 21 个字母。此时不知道下面的数字和上面的字母的对应关系，有一个电池、一个灯泡和许多很短的电线，如何只上下楼一次就能确定电线线头的对应关系？

在下面把 2 和 3 连在一起，把 4~6 全连在一起，把 7~10 全连在一起……这样就把电线分成了 6 个“等价类”，大小分别为 1, 2, 3, 4, 5, 6。然后爬到楼顶，测出哪根线和其他所有电线都不相连，哪些线和另外一根相连，哪些线和另外两根相连等，从而确定出字母 A~U 各属于哪个等价类。

然后把每个等价类中的第一个字母连在一起，形成一个大小为 6 的新等价类；再把后 5 个等价类中的第二个字母连在一起，形成一个大小为 5 的新等价类；以此类推。再回到楼下，把新的等价类区别出来。此时就知道了每个数字对应了哪一个原等价类的第几个字母，从而解决问题。

3. 蚂蚁相撞问题

题目：在一个等边三角形的 3 个顶点上各有一只蚂蚁，它们向另一个顶点运动，目标随机（可能为另外两个顶点的任意一个）。问 3 只蚂蚁不相撞的概率是多少？

1/4。

先取 3 只蚂蚁中的任意一只做研究，它的行动路线可以向另外两个顶点的任意一个移动，然后取第二只蚂蚁，为了要使 3 只蚂蚁互不相撞，它必须不能与第一只蚂蚁相向而行，所以只有 1 种行动路线，而它总共有两条线路可供选择，所以它们互不相撞的可能性是 1/2。最后取第 3 只蚂蚁，前面两只蚂蚁的路线都确定好以后，它只能从可选的两条路里面走唯一一条使它们互不相撞的路线，也就是 3 个蚂蚁做相同方向的绕圈运动，而第三只蚂蚁为了使它们互不相撞，选择路线的可能性也是 1/2。所以 3 只蚂蚁不相撞的概率是 $1/2 \times 1/2 = 1/4$ 。

还可以换一种思维来进行，以二进制中的 0 和 1 来表示蚂蚁的爬行方向，蚂蚁顺时针爬行记为 0，逆时针爬行记为 1，那么 3 只蚂蚁的状态可能为 000, 001, ..., 110, 111 中的任意一个，而且每种状态的概率相等，而在这 8 种状态中，只有 000 和 111 表示可以避免相撞，所以蚂蚁不相撞的概率为 1/4。

4. 小老鼠与毒酒问题

题目：有 1000 桶酒，其中只有 1 桶有毒，而一旦喝了有毒的酒，毒性就会在 1 周后发作。现在用小老鼠做实验，要在 1 周内找出那桶毒酒，问最少需要多少只小老鼠。

10 只。因为 2^{10} 等于 1024，所以 10 只小老鼠最多可以测 1024 桶酒。

先假设有 1024 个瓶子，其中只有 1 瓶毒药。

(1) 将 1024 个瓶子分成两个 512，即 512a 和 512b。从 512a 的各瓶中，各取 1 滴水，给 1 号小老鼠吃。

(2) 将两个 512 分别分成两个 256，即 512a 分成了 256a 和 256b，并且 512b 也分成了 256a、256b。从两个 256a 中，照旧每瓶取一滴，给 2 号小老鼠吃。

(3) 同样的道理，依次分为 4 个 128a、128b，将 a 各取一滴，给 3 号小老鼠吃。

(4) 8 个 64a、64b，将 a 各取一滴，给 4 号小老鼠吃。

(5) 16 个 32a、32b，将 a 各取一滴，给 5 号小老鼠吃。

(6) 32 个 16a、16b，将 a 各取一滴，给 6 号小老鼠吃。

(7) 64 个 8a、8b，将 a 各取一滴，给 7 号小老鼠吃。

(8) 128 个 4a、4b, 将 a 各取一滴, 给 8 号小老鼠吃。

(9) 256 个 2a、2b, 将 a 各取一滴, 给 9 号小老鼠吃。

(10) 512 个 1a、1b, 将 a 各取一滴, 给 10 号小老鼠吃。

然后, 经过一周的等待, 则可以得出如下结论:

(1) 如果 1 号小老鼠死, 则毒药在 512a 中; 否则, 在 512b 中。

(2) 如果 2 号小老鼠死, 则毒药在 256a 中; 否则, 在 256b 中。同时, 根据 1 的结果, 可判定这个 256 来自 512a 还是 512b。

以此类推, 可以唯一地确定这个“1”来自哪里, 也就确定了它是第几瓶。

除了以上这种方法外, 还可以采用另外一种方法, 就是二进制表示的方法。首先, 将酒编号为 1~1000 号, 然后将 10 只小老鼠分别编号为 1、2、4、8、16、32、64、128、256、512。

给小老鼠喂酒时, 让酒的编号等于小老鼠编号的加和。例如, 17 号酒喂给 1 号和 16 号小老鼠, 76 号酒喂给 4 号、8 号和 64 号小老鼠, 七天后将死掉的小老鼠编号加起来, 得到的编号就是有毒的那桶酒。因为对于任何一个小于 1024 的数, 都可以采用前面的唯一一组二进制数 (例如: 01, 10, 100, 1000, ..., 1000000000) 来表示, 所以结论成立。

5. 糖水问题

题目: 有 5 杯水, 其中有一杯是糖水, 再给你一个空杯子, 设计一种方案最多只尝 3 次, 找出这杯糖水。

此题可以使用类似于二分查找的方法进行解答。首先选取其中的 3 杯水, 都倒一点到空杯中, 搅拌均匀, 然后尝一下, 此时杯中水的味道可能出现两种情况: 有甜味与无甜味。

如果杯中的水有甜味, 则表明这 3 杯水中必有一杯是糖水, 此时从这 3 杯水中选两杯, 都倒一点到空杯中, 搅拌均匀, 然后尝一下, 如果没有甜味, 那么没选中的那一杯就是糖水, 如果有甜味, 则品尝其中的一杯水就知道哪杯是糖水了。

如果杯中水没有甜味, 那就尝一下没有选中的两杯水中的一杯, 此时就知道哪杯是糖水了。

6. 握手问题

题目: 一对夫妇邀请 $N-1$ 对夫妇参加聚会 (因此聚会上总共有 $2N$ 个人)。每个人都和所有自己不认识的人握了一次手。然后, 男主人问其余所有人 (共 $2N-1$ 个人) 各自都握了几次手, 得到的答案全部都不一样。假设每个人都认识自己的配偶, 那么女主人握了几次手?

由于聚会上一共有 $2N$ 个人, 每个人都和所有自己不认识的人握了一次手, 所以女主人握手次数只可能是从 $0 \sim 2N-2$ 这 $2N-1$ 个数。除去男主人外, 一共有 $2N-1$ 个人, 因此每个数恰好出现了一次。其中有一个人 (0) 没有握手, 即握手次数为 0, 有一个人 ($2N-2$) 和所有其他的夫妇都握了手, 即握手次数为 $2N-2$, 而这两个人肯定是一对夫妻, 否则后者将和前者握手 (从而前者的握手次数不再是 0)。除去这对夫妻外, 有一个人 (1) 只与 ($2N-2$) 个人握过手, 有一个人 ($2N-3$) 和除了 (0) 以外的其他夫妇都握了手, 这两个人肯定是一对夫妻, 否则后者将和前者握手 (从而前者的握手次数不再是 1)。依此类推, 直到握过 $N-2$ 次手的人和握过 N 次手的人配成一对。此时, 除了男主人及其配偶以外, 其余所有人都已经配对。根据排除法, 最后剩下的那个握手次数为 $N-1$ 的人就是女主人了。

7. 飞船处理器问题

题目: 一个飞船上的计算机有 n 个处理器。突然, 飞船遭遇意外, 一些处理器被损坏了。此时知道有超过一半的处理器仍然是好的, 同时可以向一个处理器询问另一个处理器是好的还是坏的, 好的处理器总是说真话, 坏的处理器总是说假话, 用 $n-2$ 次询问找出一个好的处理器。

首先给处理器从 1~n 进行编号。用符号 $a \rightarrow b$ 表示向标号为 a 的处理器询问处理器 b 是不是好的。

然后执行 $1 \rightarrow 2$ ，如果 1 说不是，就把他们俩都去掉，因为如果 1 是好的，则 2 是坏的；如果 1 是坏的，则 2 是好的，所以能够保证两个处理器一个是好的，一个是坏的，去掉它们两，则剩下的处理器中好的仍然过半，然后从 $3 \rightarrow 4$ 开始继续发问。如果 1 说 2 是好的，就继续问 $2 \rightarrow 3$ ， $3 \rightarrow 4$...直到某一次 j 说 j+1 是坏的，把 j 和 j+1 去掉，然后问 $j-1 \rightarrow j+2$ ；或者从 $j+2 \rightarrow j+3$ 开始发问，如果前面已经没有 j-1 了（之前已经被去掉了）。注意到整个推理过程，始终维护着一个类似于“链”的结构，即前面的每一个处理器都说后面那个是好的。这条链里的所有处理器要么都是好的，要么都是坏的。当这条链越来越长，剩下的处理器越来越少时，总有一个时候这条链超过了剩下的处理器的一半，此时可以肯定这条链里的所有处理器都是好的。或者，越来越多的处理器都被去掉了，链的长度依旧为 0，而最后只剩下一个或两个处理器没被问过，那它们一定就是好的了。另外注意到，第一个处理器的好坏从来没被问过，因为最后一个处理器的好坏不可能被问到，一旦链长超过剩余处理器的一半，或者最后没被去掉的就只剩这一个时，就不需要问了，因此询问次数不会超过 $n-2$ 。

8. 机器人相遇问题

题目：有两个机器人，初始时位于数轴上的不同位置，如何给这两个机器人输入一段相同的程序，使得这两个机器人保证可以相遇。注意，程序只能包含“左移 n 个单位”、“右移 n 个单位”，条件判断语句 if，循环语句 while，以及两个返回 Boolean 值的函数“在自己的起点处”和“在对方的起点处”，不能使用其他的变量和计数器。

为了保证两个机器人可以相遇，刚开始可以将两个机器人同时以单位速度右移，直到一个机器人走到另外一个机器人的起点处，然后该机器人以双倍速度追赶对方，这样两个机器人必能相遇。原理如下：假设两个机器人相距 x 个单元，标记位于左边的机器人为 A，位于右边的机器人为 B，当 A 向右移动 x 个单元到达 B 的起点处时，此时 B 也向右移动了 x 个单元，然后 A 以两倍的速度追赶元素 B，假设花费 t 时间追赶上 B，则满足等式： $2 \times t = 1 \times t + x$ ，则 $t = x$ ，则在距离 B 为 2x 的位置，A 追赶上 B。

6.9 过桥类

1. 黑夜过桥

题目：小明一家人过一座桥，过桥时是黑夜，所以必须有灯。现在小明过桥要 1 秒，小明的弟弟要 3 秒，小明的爸爸要 6 秒，小明的妈妈要 8 秒，小明的爷爷要 12 秒。每次此桥最多可过两人，而过桥的速度依过桥最慢者而定，而且灯在点燃后 30 秒就会熄灭。问：小明一家如何过桥，才能在灭灯前到达对岸？

由于每次只能过两人，为了使小明一家在有限的时间内顺利地过桥，需要考虑到时间的搭配问题，具体步骤如下：

- (1) 小明和小明的弟弟先过去，用时 3 秒，此时小明和小明的弟弟已经在对岸。
- (2) 小明的弟弟单独一个人过桥回来，用时 3 秒，而此时小明在对岸。
- (3) 小明的妈妈和小明的爷爷过去，用时 12 秒，此时小明、小明的妈妈以及小明的爷爷 3 个人在河的对岸。
- (4) 小明独自一个人过桥返回，用时 1 秒，此时小明的妈妈、小明的爷爷在对岸。
- (5) 小明和小明的爸爸一起过桥，用时 6 秒，此时小明、小明的爸爸、小明的爷爷、小明

的妈妈在河的对岸。

(6) 小明独自一人回来, 用时 1 秒, 此时小明的爸爸、小明的爷爷以及小明的妈妈在河对岸。

(7) 小明和小明弟弟过去, 用时 3 秒, 此时全家人都已经顺利到达对岸。

整个过程一共耗费时间为 $3+3+12+1+6+1+3=29$ (秒), 满足题目要求的 30 秒过河。

2. 猎人与熊

题目: 3 个猎人带着一只黑熊和两只棕熊过河, 而船很小, 每次只能载两个人或者两只熊, 或者一个人一只熊过河, 3 个猎人都划船, 黑熊是猎人训练过的, 也会划船, 但熊的数量一旦超过人的数量, 熊就会吃人, 如何进行调度, 3 个猎人才可以带着 3 只熊顺利过河?

第一个猎人先带一只棕熊过河, 然后第一个猎人独自返回来, 带第二只黑熊过河, 然后第一个猎人再独自返回来, 带上第二个猎人过河, 过河之后第一个猎人把上次带过去的黑熊带回走, 带回去之后, 再带第三个猎人过河, 然后第一个猎人再返回去, 带黑熊过河, 然后第一个猎人再返回带走最后一只黑熊, 这样就将 3 只熊顺利地带到了河对岸, 而且也保证了三个猎人没有一个会被吃掉。

6.10 概率类

1. 选弹球

题目: 有两个罐子, 一共有 50 个红色弹球和 50 个蓝色弹球, 随机选出一个罐子, 随机选出一个弹球放入罐子, 想一种办法, 如何分配两个罐子中球的颜色与个数才能使红色弹球被选中的机会最大?

一个罐子放一个红球, 另一个罐子放 49 个红球和 50 个蓝球。

如果选中只放红球的罐子, 则必然选中红球, 此时的概率为 $1/2$, 如果选中另外一只罐子, 则此时选中红球的概率为 $49/99 \times 2$, 根据概率求和, 得到选中红球的概率为 $1/2 + 49/99 \times 2 = 74/99$, 接近 75%, 这是所能达到的最大概率了。

2. 抓果冻

题目: 有一桶果冻, 其中有黄色、绿色、红色 3 种, 闭上眼睛抓取同种颜色的两个。抓取多少个就可以确定肯定有两个同一颜色的果冻?

4 个。

此题与鸽巢原理类似。鸽巢原理的原理是: 把多于 n 个的物体放到 n 个抽屉里, 则至少有一个抽屉里的东西不少于两件。在本题中, 因为只有黄色、绿色、红色 3 种颜色的果冻, 所以抓第一次就只有 1 种颜色的果冻, 抓第二次可以有两种颜色的果冻, 也可以是两个同色 (不能肯定) 的果冻, 抓第三次可以是 3 种颜色 当然也有可能有两个或者 3 个同样颜色 (还是不能肯定) 的果冻, 只有到抓第 4 个的时候不管是什么颜色, 哪怕前三次抓到的果冻的颜色都不同, 3 种果冻各有一个, 3 个果冻是 3 种颜色 这时候肯定至少有两个果冻是一样的颜色。

下篇

面试笔试技术攻克篇

第 7 章 程序设计基础

第 8 章 数据库

第 9 章 网络与通信

第 10 章 操作系统

第 11 章 软件工程

第 12 章 发散思维

第 13 章 数据结构与算法

第 14 章 海量数据处理

程序设计基础

第 7 章

计算机的普及以及互联网的发展，使得编程已经不再神秘，而生活中的点点滴滴都离不开计算机程序，上至航空、航天、航海尖端科技，下到互联网、手机、汽车、家电等民生用品，可以说，没有程序就没有现代化的生活。计算机程序已经“飞入寻常百姓家”，会 Coding 已经不再是程序员的专利，无论是工科学生，还是理科学生，甚至是文科学生，只要掌握一定的编程基础知识，勤加练习，都可以从事 IT 研发工作，成为一名出色的程序员。

7.1 C/C++关键字

虽然说没有最强大的语言，只有最强大的程序员，但是语言毕竟是编程的基础，掌握基本的语言知识是编程的前提条件。关键字是组成语言的最基本单位，对关键字的理解，有助于编写高质量的代码。

7.1.1 static（静态）变量有什么作用

在 C 语言中，关键字 `static` 的意思是静态，它有 3 个明显的作用：1) 在函数体内，静态变量具有“记忆”功能，即一个被声明为静态的变量在这一函数被调用的过程中其值维持不变。2) 在模块内（但在函数体外），它的作用域范围是有限制的，即如果一个变量被声明为静态的，那么该变量可以被模块内所有函数访问，但不能被模块外其他函数访问。它是一个本地的全局变量，如果一个函数被声明为静态的，那么该函数与普通函数作用域不同，其作用域仅在本文件中，它只可被这一模块内的其他函数调用，不能被模块外的其他函数调用，也就是说这个函数被限制在声明它的模块的本地范围内使用。3) 内部函数应该在当前源文件中说明和定义，对于可在当前源文件以外使用的函数，应该在一个头文件中说明，使用这些函数的源文件要包含这个头文件。

具体而言，`static` 全局变量和普通的全局变量的区别在于 `static` 全局变量只初始化一次，这样做的目的是为了防止在其他文件单元中被引用。`static` 局部变量和普通局部变量的区别在于 `static` 局部变量只被初始化一次，下一次的运算依据是上一次的结果值。`static()` 函数与普通函数的区别在于作用域不一样，`static()` 函数只在一个源文件中有效，不能被其他源文件使用。

在 C++ 中，在类内数据成员的声明前加上关键字 `static`，该数据成员就是类内的静态数据成员。静态数据成员有以下特点：

(1) 对于非静态数据成员，每个类对象都有自己的复制品。而静态数据成员被当做是类的成员。无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份复制品，由该类型的所有对象共享访问。

(2) 静态数据成员存储在全局数据区。定义时要分配空间，所以不能在类声明中定义。由于静态数据成员属于本类的所有对象共享，所以它不属于特定的类对象，在没有产生类对象时其作用域就可见，即在没有产生类的实例时，程序员也可以使用它。

(3) 静态数据成员和普通数据成员一样遵从 `public`、`protected`、`private` 访问规则。

(4) `static` 成员变量的初始化是在类外，此时不能再带上 `static` 的关键字 `private`、`protected` 的 `static` 成员虽然可以在类外初始化，但是不能在类外被访问。

与全局变量相比，使用静态数据成员有以下两个优势：

(1) 静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其他全局名字冲突的可能性。

(2) 可以实现信息隐藏。静态数据成员可以是 `private` 成员，而全局变量不能。

需要注意的是，类的静态成员必须初始化，因为它是在程序初始化的时候分配的。类中只是声明，在 `cpp` 中才是初始化，可以在初始化的代码上放个断点，在程序执行 `main()` 的第一条语句之前就会先走到那儿。如果静态成员是个类，那么就会调用到它的构造函数。

与静态数据成员一样，当类的成员函数前面添加了 `static` 关键字后就变为了类的静态成员函数，静态成员函数为类的全部服务而不是为某一个类的具体对象服务。静态成员函数是类的内部实现，属于类定义的一部分。普通的成员函数一般都隐含了一个 `this` 指针，`this` 指针指向类的对象本身，因为普通成员函数总是具体的属于某个类的具体对象的。通常情况下，`this` 是默认的。如函数 `fn()` 实际上是 `this->fn()`。但是与普通函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有 `this` 指针。从这个意义上讲，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数。

引申 1：为什么 `static` 变量只初始化一次？

对于所有的对象（不仅仅是静态对象），初始化都只有一次，而由于静态变量具有“记忆”功能，初始化后，一直都没有被销毁，都会保存在内存区域中，所以不会再次初始化。

存放在静态区的变量的生命周期一般比较长，一般与整个源程序“同生死、共存亡”，所以它只需初始化一次。而 `auto` 变量，即自动变量，由于存放在栈区，一旦调用过程结束，就会立刻被销毁。

分析以下程序代码：

```
#include<stdio.h>

void fun(int i)
{
    static int value=i++;
    printf("%d\n",value);
}

int main( )
{
    fun(0);
    fun(1);
    fun(2);
    return 0;
}
```

程序输出为

```
0
0
0
```

程序每次输出都为 0，是因为 `value` 是静态类型（`static`），只会定义一次。也就是说，不管调用 `fun()` 这个函数多少次，`static int value = i++` 这个定义语句只会在第一次调用的时候执行，由于第一次执行的时候 `i = 0`，所以 `value` 也就被初始化成 0 了，以后调用 `fun()` 都不会

再执行这条语句的。

分析以下一段代码：

```
#include<stdio.h>

void fun(int i)
{
    static int value=i++;
    value = i++;
    printf("%d\n",value);
}

int main( )
{
    fun(0);
    fun(1);
    fun(2);
    return 0;
}
```

程序输出为

```
1
1
2
```

上述代码之所以输出为 1,1,2，是因为当调用 fun(0)时，由于 value 被声明为 static，所以定义语句只执行一次，此时 value=i++，value 的值为 0，i 的值变为 1，执行第二行语句 value=i++后，此时 value 的值为 i 的初值为 1，接着 i 的值变为 2，所以第一次输出为 1。当调用 fun(1)时，因为 value 是静态变量，具有记忆功能，所以会跳过定义语句，只执行 value=i++语句，所以 value 的值为 1，而此时 i 的值变为 2，所以第二次调用时输出为 1。当调用 fun(2)的时候，也会跳过定义语句，只执行 value=i++语句，所以 value 的值为 2，i 的值变为 3，所以第三次调用时输出为 2。

引申 2：在头文件中定义静态变量，是否可行？为什么？

不可行，如果在头文件中定义静态变量，会造成资源浪费的问题，同时也可能引起程序错误。因为如果在使用了该头文件的每个 C 语言文件中定义静态变量，按照编译的步骤，在每个头文件中都会单独存在一个静态变量，从而会引起空间浪费或者程序错误。

所以不推荐在头文件中定义任何变量，当然也包括静态变量。

7.1.2 const 有哪些作用

常类型也称为 const 类型，是指使用类型修饰符 const 说明的类型。const 是 C 和 C++中常见的关键字，在 C 语言中，它主要用于定义变量为常类型以及修饰函数参数与返回值，而在 C++中还可以修饰函数的定义，定义类的成员函数。常类型的变量或对象的值是不能被更新的。

一般而言，const 有以下几个方面的作用：

(1) 定义 const 常量，具有不可变性。例如：

```
const int MAX=100;
int Array[MAX];
```

(2) 进行类型检查，使编译器对处理内容有更多的了解，消除了一些隐患。例如：void f(const int i) { ... } 编译器就会知道 i 是一个常量，不允许修改。

(3) 避免意义模糊的数字出现，同样可以很方便地进行参数的调整和修改。同宏定义一

样,可以做到不变则已,一变都变。如(1)中,如果想修改 MAX 的内容,只需要定义 `const int MAX=期望值` 即可。

(4) 保护被修饰的东西,防止被意外的修改,增强了程序的健壮性。在上例中,如果在函数体内修改了变量 `i` 的值,那么编译器就会报错。例如:

```
void f(const int i)
{
    i=10;
}
```

上述代码对 `i` 赋值会导致编译错误。

(5) 为函数重载提供参考。

```
class A
{
    void f(int i) {...} //定义一个函数
    void f(int i) const {...} //上一个函数的重载
}
```

(6) 节省空间,避免不必要的内存分配。例如:

```
#define PI 3.14159 //该宏用来定义常量
const double Pi=3.14159; //此时并未将 Pi 放入只读存储器中
double i=Pi; //此时为 Pi 分配内存,以后不再分配
double I=Pi; //编译期间进行宏替换,分配内存
double j=Pi; //没有内存分配
double J=Pi; //再次进行宏替换,又一次分配内存
```

`const` 定义常量从汇编的角度来看,只是给出了对应的内存地址,而不是像 `#define` 一样给出的是立即数,所以 `const` 定义的常量在程序运行过程中只有一份复制品,而 `#define` 定义的常量在内存中有若干个复制品。

(7) 提高了程序的效率。编译器通常不为普通 `const` 常量分配存储空间,而是将它们保存在符号表中,这使得它成为一个编译期间的常量,没有了存储与读内存的操作,使得它的效率也很高。

引申 1: 什么情况下需要使用 `const` 关键字?

(1) 修饰一般常量。一般常量是指简单类型的常量。这种常量在定义时,修饰符 `const` 可以用在类型说明符前,也可以用在类型说明符后。例如: `int const x=2` 或 `const int x=2`。

(2) 修饰常数组。定义或说明一个常数组可以采用如下格式:

```
int const a[8]={1, 2, 3, 4, 5, 6, 7, 8};
const int a[8]={1, 2, 3, 4, 5, 6, 7, 8};
```

(3) 修饰常对象。常对象是指对象常量,定义格式如下:

```
class A;
const A a;
A const a;
```

定义常对象时,同样要进行初始化,并且该对象不能再被更新,修饰符 `const` 可以放在类名后面,也可以放在类名前面。

(4) 修饰常指针。

```
const int *A; //const 修饰指向的对象, A 可变, A 指向的对象不可变
int const *A; //const 修饰指向的对象, A 可变, A 指向的对象不可变
int *const A; //const 修饰指针 A, A 不可变, A 指向的对象可变
const int *const A; //指针 A 和 A 指向的对象都不可变
```

(5) 修饰常引用。使用 `const` 修饰符也可以说明引用,被说明的引用为常引用,该引用所引用的对象不能被更新。其定义格式如下:

```
const double & v;
```

(6) 修饰函数的常参数。`const` 修饰符也可以修饰函数的传递参数，格式如下：

```
void Fun(const int Var);
```

告诉编译器 `var` 在函数体中的无法改变，从而防止了使用者一些无意的或错误的修改。

(7) 修饰函数的返回值。`const` 修饰符也可以修饰函数的返回值，返回值不可被改变，格式如下：

```
const int Fun1( );
const MyClass Fun2( );
```

(8) 修饰类的成员函数。`const` 修饰符也可以修饰类的成员函数，格式如下：

```
class ClassName
{
    public:
    int Fun( ) const;
};
```

这样，在调用函数 `Fun()` 时就不能修改类或对象的属性。

(9) 在另一连接文件中引用 `const` 常量。使用方式有：

```
extern const int i;
extern const int j=10;
```

第一种用法是正确的，而第二种用法是错误的，常量不可以被再次赋值。另外，还要注意，常量必须初始化，如 `const int i=5`。

引申 2：什么是常引用？

常引用也称为 `const` 引用。之所以引入常引用，是为了避免在使用变量的引用时，在毫不知情的情况下改变了变量的值，从而引起程序错误。常引用主要用于定义一个普通变量的只读属性的别名，作为函数的传入形参，避免实参在调用函数中被意外地改变。

`const` 引用的意思是指向 `const` 对象的引用，非 `const` 引用表示指向非 `const` 类型的引用。如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。常引用声明方式：

```
const 类型标识符 & 引用名 = 目标变量名;
```

常引用的主要用途如下：

(1) 用做普通变量的只读属性的别名。通常这个别名只能获得这个变量的值，而不能改变这个变量的值。

(2) 用于函数的形参。常引用做形参，可以确保在函数内不会改变实参的值，所以参数传递时尽量使用常引用类型。

如果是对一个常量进行引用，则编译器首先建立一个临时变量，然后将该变量的值置入临时变量中，对该引用的操作就是对该临时变量的操作，对常量的引用可以用其他任何引用来初始化，但不能改变。

关于引用的初始化，一般需要注意以下问题：当初始化值是一个左值（可以取得地址）时，没有任何问题；而当初始化值不是一个左值时，则只能对一个常引用赋值，而且这个赋值是有一个过程的，首先将值隐式转换到类型 `T`，然后将这个转换结果存放在一个临时对象里，最后用这个临时对象来初始化这个引用变量。例如如下两种使用方式：

```
(1) double& dr = 1;
```

```
(2) const double& cdr = 1;
```

第 (1) 种方法错误，初始化值不是左值，而第 (2) 种方法正确，其对应的执行过程如下：

```
double temp = double(1);
```

```
const double& cdr = temp;
```

如果对第(1)种使用方法进行相应的改造,也可以变为合法,例如:

```
const int ival = 1024;
```

```
(1) const int &refVal=ival;
```

```
(2) int &ref2=ival;
```

在上例中,第(1)种方法的引用是合法的,而第(2)种方法的引用是非法的。上例中,可以读取 refVal 的值,但是不能修改它,因为 refVal 的类型是 const,任何对 refVal 的赋值都是不合法的(const 引用是只读的,常量即不能作为左值的量,定义式中赋初值除外)。同时,const 引用可以初始化为不同类型的对象或者初始化为右值,如字面值常量,而非 const 引用只能绑定到与该引用同类型的对象。例如,下述 const 引用都是合法的。

```
int i = 42;
```

```
const int &r = 42;
```

```
const int &r2 = r + i;
```

在使用 const 引用进行函数调用的时候,需要注意一个问题,例如如下函数声明:

```
void bar(string & s);
```

那么下面的表达式将是非法的:

```
bar("hello world");
```

程序示例如下所示:

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
void bar(const string& s)
```

```
{
```

```
    cout<<s<<endl;
```

```
}
```

```
int main( )
```

```
{
```

```
    bar("hello world");
```

```
    return 0;
```

```
}
```

程序输出为

```
hello world
```

原因在于“hello world”串会产生一个临时对象,而在 C++中,临时对象是 const 类型的。因此上面的表达式就是试图将一个 const 类型的对象转换为非 const 类型,这是非法的。引用型参数应该在能被定义为 const 的情况下,尽量定义为 const。

7.1.3 switch 语句中的 case 结尾是否必须添加 break 语句? 为什么

一般必须在 case 语句结尾添加 break 语句。因为一旦通过 switch 语句确定了入口点,所有进一步的 case 语句都会被忽略,并且除非遇到关键字 break,否则会执行满足这个 case 之后的其他 case 的语句,直到 switch 结束或者遇到 break 为止。如果在 switch 中省略了 break 语句,那么匹配的 case 值后的所有情况(包括 default 情况)都会被执行。

程序代码如下所示:

```
#include <stdio.h>
```

```
int main( )
```

```
{
```



```

int i;
for(i=0;i<3;i++)
{
    switch (i)
    {
        case 0:
            printf("%d\n",i);
        case 2:
            printf("%d\n",i);
        default :
            printf("%d\n",i);
    }
}
return 0;
}

```

输出为:

```

0
0
0
1
2
2

```

case 0 的时候执行 3 次打印, case 1 的时候执行一次 default, case 2 的时候执行两次打印。如果将 case 2 后面添加 break 语句, 则最后输出为 0012, 因为此时 case 0 执行两次, case 1 执行一次 default, case 2 执行一次。

需要注意的是, switch(c) 语句中 c 可以是 int、long、char、unsigned int 等类型, 唯独不可以是 float 类型。

7.1.4 volatile 在程序设计中有何作用

编译器优化的时候可能会出现一些问题, 如当遇到多线程编程时, 变量的值可能因为别的线程而改变了, 而该寄存器的值不会相应改变, 从而造成应用程序读取的值和实际的变量值不一致。例如, 在本次线程内, 当读取一个变量时, 为提高存取速度, 编译器优化时有时会先把变量读取到一个寄存器中; 当以后再取变量值时, 就直接从寄存器中取值; 当变量值在本线程里改变时, 会同时把变量的新值复制到该寄存器中, 以便保持一致。

volatile 是一个类型修饰符 (type specifier), 它用来修饰被不同线程访问和修改的变量。被 volatile 类型定义的变量, 系统每次用到它的时候都是直接从对应的内存当中提取, 而不会利用 cache 中的原有数值, 以适应它的未知何时会发生的变化, 系统对这种变量的处理不会做优化。所以, volatile 一般用于修饰多线程间被多个任务共享的变量和并行设备硬件寄存器等。

对于 volatile 关键字的作用, 可以通过在代码中插入汇编代码, 测试有无 volatile 关键字对程序最终代码的影响。

首先建立一个 voltest.cpp 文件, 输入下面的代码:

```

#include <stdio.h>
int main( )
{
    int i=10;
    int a = i;
    printf("i= %d\n",a); //下面汇编语句的作用是改变内存中 i 的值, 但是又不让编译器知道
    _asm

```

```

    {
        mov    dword ptr [ebp-4], 20h
    }
    int b = i;
    printf("i= %d\n",b);
    return 0;
}

```

在 debug 调试版本模式运行程序，输出结果如下：

```

i = 10
i = 32

```

在 release 版本模式运行程序，输出结果如下：

```

i = 10
i = 10

```

输出的结果明显表明，在 release 模式下，编译器对代码进行了优化，第二次没有输出正确的 i 值。把 i 的声明加上 volatile 关键字，程序实例如下：

```

#include <stdio.h>
int main( )
{
    volatile int i=10;
    int a = i;
    printf("i= %d\n",a);//下面汇编语句的作用是改变内存中 i 的值，但是又不让编译器知道
    _asm
    {
        mov    dword ptr [ebp-4], 20h
    }
    int b = i;
    printf("i= %d\n",b);
    return 0;
}

```

分别在 debug 调试版本和 release 发布版本运行程序，输出如下所示：

```

i = 10
i = 32

```

一个定义为 volatile 的变量是说这个变量可能会被意想不到地改变，这样编译器就不会去假设这个变量的值了。准确地说，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值（From Memory），而不是使用保存在寄存器里的备份。

7.1.5 断言 ASSERT()是什么

ASSERT()一般被称为断言，它是一个调试程序时经常使用的宏。它定义在<assert.h>头文件中，通常用于判断程序中是否出现了非法的数据，在程序运行时它计算括号内的表达式的值。如果表达式的值为 false(0)，程序报告错误，终止运行，以免导致严重后果，同时也便于查找错误；如果表达式的值不为 0，则继续执行后面语句。在此需要强调一点，ASSERT()捕获的是非法情况，而非错误情况，错误情况是必然存在的，并且一定需要作出相应的处理，而非法情况则不是，它可能只是漏洞而已。

其用法如下：

```

ASSERT (n!=0);
k=10/n;

```

需要注意的是，ASSERT()只在 Debug 版本中有，编译的 Release 版本则被忽略。还需要注意的一个问题是 ASSERT()与 assert()的区别，ASSERT()是宏，而 assert()是 ANSI C 标准中规定的函数，它与 ASSERT()的功能类似，但是可以应用在 Release 版本中。

使用 `assert()` 的缺点是，频繁的调用会极大地影响程序的性能，增加额外的开销。在调试结束后，可以通过在包含 `#include <assert.h>` 的语句之前插入 `#define NDEBUG` 来禁用 `assert()` 调用，示例代码如下：

```
#include <stdio.h>
#define NDEBUG
#include <assert.h>
```

对于 `assert()` 的使用，需要注意以下几个方面：

(1) 在函数开始处检验传入参数的合法性。例如：

```
assert(nNewSize >= 0);
assert(nNewSize <= MAX_BUFFER_SIZE);
```

(2) 每个 `assert()` 一般只检验一个条件，而不对多个条件进行检验，因为同时检验多个条件时，如果断言失败，则无法直观地判断是哪个条件失败。例如，`assert(nOffset>=0 && nOffset+nSize<=m_nInfomationSize)` 就不是一种高效的方式，它无法判断是 `nOffset>=0` 有误还是 `nOffset+nSize<=m_nInfomationSize` 有误，而将该语句分开表示为如下两个简单语句则更好：`assert(nOffset >= 0)` 和 `assert(nOffset+nSize <= m_nInfomationSize)`。

(3) 不能使用改变环境的语句，因为 `assert` 只在 `DEBUG` 时生效，如果这么做，会使程序在真正运行时遇到问题。例如，`assert(i++<100)` 就是错误的。如果执行出错，在执行之前 `i=100`，那么这条语句就不会执行，`i++` 这条命令就没有执行。而正确的写法应该为 `assert(i<100);i++`。

(4) 并非所有的 `assert()` 都能代替过滤条件，对于有的地方，`assert()` 无法达到条件过滤的目的。

(5) 一般在编程的时候，为了形成逻辑和视觉上的一致性，会将 `assert()` 与后面的语句之间空一行来隔开。

7.1.6 枚举变量的值如何计算

以如下程序实例进行分析。

```
#include<stdio.h>
```

```
int main( )
```

```
{
```

```
    enum {a,b=5,c,d=4,e};
```

```
    printf("%d %d %d %d %d\n",a,b,c,d,e);
```

```
    return 0;
```

```
}
```

程序输出为

```
0 5 6 4 5
```

为什么 `c` 的值为 6 呢？其实，在枚举中，某个枚举变量的值默认为前一个变量的值加 1，而如果第一个枚举变量没有被赋值，则其默认值为 0。所以在上例中，`a`，`b`，`c`，`d`，`e` 的值分别为 0，5，6，4，5，其中 `b` 与 `e` 的值都为 5，从这个例子中还可以看出枚举变量值是可以重复的。

7.1.7 `char str1[] = "abc"; char str2[] = "abc"; str1 与 str2 不相等，为什么`

两者不相等，是因为 `str1` 和 `str2` 都是字符数组，每个都有其自己的存储区，它们的值则是各存储区的首地址。但有些情况却不一样，程序示例如下：

```
#include<iostream>
using namespace std;

int main( )
{
    const char str3[] = "abc";
    const char str4[] = "abc";
    const char* str5 = "abc";
    const char* str6 = "abc";
    cout << boolalpha << ( str3==str4 ) << endl;
    cout << boolalpha << ( str5==str6 ) << endl;
    return 0;
}
```

程序输出为

```
false
true
```

为什么上面程序示例的输出结果不都是 false 呢？上例中，str3 和 str4 两个字符数组都存储在栈空间上，但两者地址值不相等。而 str5 和 str6 并非字符数组而是字符指针，并不分配存储区，其后的“abc”以常量形式存于常量区，str5 和 str6 是指它们指向的地址的首地址，而它们自己仅是指向该区首地址的指针，所以相等（&str5 和 &str6 是指指针自己的地址，所以两者地址是不相等的）。

7.1.8 为什么有时候 main() 函数会带参数？参数 argc 与 argv 的含义是什么

C 语言的设计原则是把函数作为程序的构成模块。在 C99 标准中，允许 main() 函数没有参数，或者有两个参数（有些实现允许更多的参数，但这只是对标准的扩展）。

命令行参数有时用来启动一个程序的执行，如 int main(int argc, char *argv[])，其中第一个参数 argc 表示命令行参数的数目，它是 int 型的；第二个参数 argv 是一个指向字符串的指针数组，由于参数的数目并没有内在的限制，所以 argv 指向这组参数值（从本质上说是一个数组）的第一个元素，这些元素的每个都是指向一个参数文本的指针。

程序代码（程序名为 a.c）如下：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int count;
    printf("该命令一共有%d 个参数:\n",argc-1);
    for(count=1;count<argc;count++)
        printf("%d: %s\n",count,argv[count]);
    return 0;
}
```

编译运行，在命令行输入 c I love you 回车，下面是运行该程序的结果：

该命令一共有 3 个参数：

```
1: I
2: love
3: you
```

在本例中，程序从命令行中接受了 4 个字符串（此处包括程序名），并将它们存储在字符串数组中，其中 argv[0] 表示 a(程序名)，argv[1] 对应字符串 I，argv[2] 对应字符串 love，argv[3] 对应字符串 you。argc 的值为参数的个数，程序自动统计。

同时需要注意的是，一个 C 语言程序总是从 main() 函数开始执行的。

7.1.9 C++里面是不是所有的动作都是 main() 函数引起的

不是，对于 C++ 程序而言，静态变量、全局变量、全局对象的分配早在 main() 函数之前已经完成。所以并不是所有的动作都是由 main() 引起的，只是编译器是由 main() 开始执行的，main() 只不过是一个约定的函数入口，在 main() 函数中的显示代码执行之前，会调用一个由编译器生成的 _main() 函数，而 _main() 函数会进行所有全局对象的构造及初始化工作。

以如下程序示例代码为例：

```
class A{};
A a;
int main()
{
    ...
}
```

程序在执行时，因为会首先初始化全局变量，当这个变量是一个对象时，则会首先调用该对象的构造函数，所以上例中，a 的构造函数先执行，然后再执行 main() 函数。C++ 中并非所有的动作都是 main() 引起的。

怎样在 main() 函数退出之后再执行一段代码？答案依然是全局对象，当程序退出时，全局变量必须销毁，自然会调用全局对象的析构函数，所以剩下的就同构造函数一样了。

7.1.10 *p++ 与 (*p)++ 等价吗？为什么

在回答这个问题前，必须弄明白一个问题，就是 C 语言中操作符的优先级问题。在 C 语言中，优先级由高到低的排序主要遵循如下规则：

(1) 函数符号()，数组下标[]，数组下标符号.，成员符号->，结合性从左往右。

(2) 单目运算符：!，~，++，--，-(type)*，&，sizeof 结合性从右往左。

(3) 算术运算符：*、/、%，结合性从左往右。

(4) +、- 结合性从左往右。

(5) 移位运算符：<<，>>，>>> 结合性从左往右。

(6) 关系运算符：<，<=，>，>= 结合性从左往右。

(7) ==，!= 结合性从左往右。

(8) 逻辑运算符：首先，按位运算符&、^与|，且&高于^，^高于|，结合性从左往右；其次，逻辑运算符&&与||，且&&高于||，结合性从左往右。

(9) 三目运算符? :，结合性从右往左；其次是赋值运算符=，结合性从右往左；最后是逗号运算符，结合性从左往右。

对于操作符的优先级总结如下：

(1) 关系运算符优于逻辑运算符。

(2) 移位运算符介于算术运算符和比较运算符之间。

(3) 除单目运算符外，算术运算符的优先级最高。

所以，因为优先级顺序的问题，*p++ 与 (*p)++ 并不等价，前者先完成取值操作，然后对指针地址执行++操作；而后者为首先执行取值操作，然后对该值进行++运算。

7.1.11 前置运算与后置运算有什么区别

以++操作为例，对于变量 a，++a 表示取 a 的地址，增加它的内容，然后把值放在寄存器中；a++ 表示取 a 的地址，把它的值装入寄存器，然后增加内存中 a 的值。

一般而言,当涉及表达式计算时,对这两种情况的计算过程区分如下:后置的++运算符是先将其值返回,然后其值增1;而前置的++运算符,则是先将值增1,再返回其值。程序实例如下:

```
#include<stdio.h>

int main( )
{
    int a,b,c,d;
    a=10;
    b=a++;
    c=++a;
    d=10*a++;
    printf("%d\n%d\n%d\n%d\n",a,b,c,d);
    return 0;
}
```

程序输出为

```
13
10
12
120
```

上例中,首先赋值 a 为 10,然后执行 $b=a++$ 语句,由于后置操作符的特性,所以首先执行 $b=a$ 操作,及 b 的值为 10,然后执行 a 的自增操作, a 的值变为 11。紧接着执行 $c=++a$ 语句,由于后置操作符的特性,所以首先执行 a 的自增操作, a 的值变为 12,然后执行 $c=a$ 这一赋值操作,所以 c 的值变为 12。当执行 $d=10*a++$ 语句时,由于++操作符的优先级大于*操作符,所以该语句等价于 $d=10*(a++)$;由于是后置操作符,所以首先执行赋值语句, d 的值变为 $10*12$,即为 120,然后 a 执行自增操作,变为 13,所以最终 a 、 b 、 c 、 d 的值分别变为 13、10、12、120。

再如:

首先定义 $\text{int } a=4$,然后分别执行以下 5 种情况:

- (1) $a += a++$;
- (2) $a += ++a$;
- (3) $++a += a$;
- (4) $++a += a++$;
- (5) $++a += ++a$;

在 VC 6.0 的环境下执行以上代码,第(1)种情况下, a 的值变为 9;第(2)种情况下, a 的值变为 10;第(3)种情况下, a 的值变为 10;第(4)种情况下, a 的值变为 11;第(5)种情况下, a 的值变为 12。

需要注意的是,对于迭代器和其他模板对象使用前缀形式($++i$)的自增、自减运算符,一般推荐使用前置自增运算,因为前置自增($++i$)通常要比后置自增($i++$)效率更高。

7.1.12 a 是变量,执行 $(a++) += a$ 语句是否合法

为了更好地说明本题,首先引入两个概念:左值和右值。左值就是可以出现在表达式左边的值(等号左边),可以被改变,它是存储数据值的那块内存的地址,也称为变量的地址;右值是指存储在某内存地址中的数据,也称为变量的数据。左值可以作为右值,但是右值不可以是左值。

本题不合法, `a++` 不能当做左值使用。`++a` 可以当左值使用。`++a` 表示取 `a` 的地址, 对它的内容进行加 1 操作, 然后把值放在寄存器中。`a++` 表示取 `a` 的地址, 把它的值装入寄存器, 然后对内存中 `a` 的值执行加 1 操作。

所以, 对于如下两种写法: (1) `i++ = 5`; (2) `++i = 5`; 第 (1) 种写法是错误的, 第 (2) 种写法是正确的。`i++` 的运算结果并不是 `i` 这个变量的引用, 而是一个临时变量, 其值为 `i` 的值, 所以无法进行 `i++=5` 运算, 甚至编译器不允许对一个临时变量重新赋值, 上面的表达式会引起编译错误。

7.1.13 如何进行 float、bool、int、指针变量与“零值”的比较

在编写程序时, 经常需要对变量与“零值”进行比较判断。考查对 0 值判断是衡量程序员基本功的重要标准, 不同变量与零值的判断, 往往方法也不一样, 但很多程序员往往会存在很多误区, 将 `NULL`、`0`、`1`、`FALSE`、`TRUE` 的意思混淆。例如, 把 `BOOL` 型变量的 0 判断可以写成 `if(var == 0)`, 把 `int` 型变量与零值比较写成 `if(!var)`, 把指针变量与零值的比较写成 `if(!var)`, 虽然上述写法程序也能正确运行, 但是未能清晰地表达程序的意思。

一般地, 如果想让 `if` 判断一个变量是真还是假, 应直接使用 `if(var)`、`if(!var)`, 表明其为“逻辑”判断; 如果用 `if` 判断一个数值型变量 (如 `short`、`int`、`long` 等), 应该用 `if(var == 0)`, 表明是与 0 进行“数值”上的比较; 而判断指针则最好使用 `if(var == NULL)`。对于浮点数的比较, 首先需要考虑到的问题就是浮点型变量在内存中的存储导致它并不是一个精确的数, 所以不可以将 `float` 变量用“`==`”或“`!=`”与数字比较, 应该设法转化成“`>=`”或“`<=`”形式。

具体而言, 分以下几种情况。

(1) `int` 类型。

```
if (n == 0)
if (n != 0)
```

不推荐的写法有:

```
if (n)
if (!n)
```

因为这样写容易让人误解 `n` 是布尔变量。

(2) `float` 类型。无论是 `float` 还是 `double` 类型的变量, 由于它们在内存中的存储机制与整型数不同, 有舍入误差, 所以在计算机中, 大多数浮点数都是无法精确表达的, 很难用 `A == B` 来判定两个浮点数是否相同。在判断浮点数相等时, 推荐用范围来确定, 若 `x` 在某一范围内, 就认为相等, 至于范围怎么定义, 要依据实际情况而定, `float` 和 `double` 也各有不同。所以都不可以用“`==`”或“`!=`”与任何数字比较, 应该设法转化成“`>=`”或“`<=`”某个精度值。具体方式如下所示:

```
const float EPSINON = 0.00001;
if ((x >= -EPSINON) && (x <= EPSINON))
```

上例中的 `EPSINON` 的值取的是 0.00001, 而一般对于该值的选取主要是按照实际情况设置的。

错误的写法有以下两种形式:

```
if (x == 0.0)
if (x != 0.0)
```

需要注意的是, 因为浮点数的精度误差, 导致对于确切的两个浮点数 `a` 与 `b`, `a+b` 的值和 `b+a` 的值永远是相等的, 而浮点数的运算是不可结合的, 所以 `(a+b)+c` 的值和 `(a+c)+b` 的值就不一定相等了。

(3) bool 类型。

```
if(flag)
if(!flag)
```

不推荐的写法有：

```
if(flag==TRUE)
if(flag==FALSE)
if(flag==1)
if(flag==0)
```

(4) 指针类型。

```
if(p == NULL)
if(p != NULL)
```

不推荐的写法有：

```
if(p == 0)
if(p != 0)
```

上述写法容易让人误解 p 是整型变量。

```
if(p)
if(!p)
```

上述写法容易让人误解 p 是 bool 型变量。

在进行比较时，有一个比较容易忽略的问题，就是将双等号“==”与单等号“=”混淆。其实与日常生活中不同的是，在计算机领域，单等号“=”表示的是赋值操作，而双等号“==”才表示比较操作。

7.1.14 new/delete 与 malloc/free 的区别是什么

在 C++ 中，申请动态内存与释放动态内存，用 new/delete 与 malloc/free 都可以，而且它们的存储方式相同，new 与 malloc 动态申请的内存都位于堆中，无法被操作系统自动回收，需要对应的 delete 与 free 来释放空间，同时对于一般的数据类型，如 int、char 型，它们的效果一样。

malloc/free 是 C/C++ 语言的标准库函数，在 C 语言中需要头文件 <stdlib.h> 的支持，new/delete 是 C++ 的运算符。对于类的对象而言，malloc/free 无法满足动态对象的要求，对象在创建的同时要自动执行构造函数，对象消亡之前要自动执行析构函数，而 malloc/free 不在编译器控制权限之内，无法执行构造函数和析构函数。

具体而言，new/delete 与 malloc/free 的区别主要表现在以下几个方面：

(1) new 能够自动计算需要分配的内存空间，而 malloc 需要手工计算字节数。例如，int* p1 = new int[2]，int* p2 = malloc(2*sizeof(int))。

(2) new 与 delete 直接带具体类型的指针，malloc 与 free 返回 void 类型的指针。

(3) new 是类型安全的，而 malloc 不是，例如，int* p = new float[2]，编译时就会报错；而 int* p = malloc(2*sizeof(int))，编译时编译器就无法指出错误来。

(4) new 一般由两步构成，分别是 new 操作和构造。new 操作对应于 malloc，但 new 操作可以重载，可以自定义内存分配策略，不做内存分配，甚至分配到非内存设备上，而 malloc 不行。

(5) new 将调用构造函数，而 malloc 不能；delete 将调用析构函数，而 free 不能。

(6) malloc/free 需要库文件 stdlib.h 支持，new/delete 则不需要库文件支持。

程序示例如下：

```
#include <iostream>
using namespace std;
```



```

class A
{
    public:
        A()
        {
            cout<<"A is here!"<<endl;
        }
        ~A()
        {
            cout<<"A is dead!"<<endl;
        }
    private:
        int i;
};

int main( )
{
    A* pA=new A;
    delete pA;
    return 0;
}

```

程序输出为

```

A is here!
A is dead!

```

需要注意的是，有资源的申请，就有资源的释放，否则就会出现资源泄露（也称内存泄露）的问题，所以 `new/delete`，`malloc/free` 必须配对使用。而且 `delete` 和 `free` 被调用后，内存不会立即收回，指针也不会指向空，`delete` 或 `free` 仅仅是告诉操作系统，这一块内存被释放了，可以用做其他用途。但是，由于没有重新对这块内存进行写操作，所以内存中的变量数值并没有发生变化，出现野指针的情况。因此，释放完内存后，应该将指针指向置位空。

程序示例如下：

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void TestFree( )
{
    char *str = (char *) malloc(100);
    strcpy(str, "hello");
    free(str);
    if(str != NULL)
    {
        strcpy(str, "world");
        printf("%s\n",str);
    }
}

int main( )
{
    TestFree( );
    return 0;
}

```

程序输出为

```

world

```

通过上例可知, `free` 或 `delete` 调用后, 内存其实并没有释放, 也没有为空, 而是还存储有内容, 所以在将资源 `free` 或 `delete` 调用后, 还需要将其置为 `NULL` 才行。

此时, 便产生了一个问题, 既然 `new/delete` 的功能完全覆盖了 `malloc/free`, 为什么在 C++ 中没有取消掉 `malloc/free`, 而是仍然将其保留了? 其实, 由于 C++ 程序经常要调用 C 函数, 而 C 程序只能用 `malloc/free` 管理动态内存, 所以仍然保留了 `malloc/free`。

7.1.15 什么时候需要将引用作为返回值

将引用作为函数返回值类型的格式如下所示:

类型标识符 &函数名 (形参列表及类型说明) { // 函数体 }

将引用作为返回值的优点是在内存中不产生被返回值的副本, 从而大大提高了程序的安全性与效率。

具体而言, 将引用作为函数返回值类型的格式一般需要注意以下 4 点内容:

(1) 不能返回局部变量的引用。局部变量由于存储在栈区, 在函数返回后会被销毁, 因此被返回的引用就成为了“无所指”的引用, 程序会进入未知状态, 引起程序错误甚至崩溃。

(2) 不能返回函数内部 `new` 分配的内存的引用。例如, 被函数返回的引用只是作为一个临时变量出现, 而没有被赋予一个实际的变量, 那么这个引用所指向的空间 (由 `new` 分配) 就无法释放, 造成内存泄露。

(3) 可以返回类成员的引用, 但最好是常引用类型。当对象的属性与某种业务规则相关联时, 其赋值常常与某些其他属性或对象的状态有关, 因此有必要将赋值操作封装在一个业务规则当中。如果其他对象可以获得该属性的非常量引用 (或指针), 那么对该属性的单纯赋值就会破坏业务规则的完整性。

(4) 流操作符 `<<` 和 `>>`。一般这两个操作符连续使用, 因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。在另外的一些操作符中, 不能返回引用 `+-*/` 四则运算符。由于这 4 个操作符没有副作用, 因此它们必须构造一个对象作为返回值, 可选的方案包括返回一个对象, 返回一个局部变量的引用, 返回一个 `new` 分配的对象引用, 返回一个静态对象引用。根据前面提到的引用作为返回值的 3 个规则, 第 (2), (3) 两个方案都被否决了。静态对象的引用又因为 `((a+b) == (c+d))` 会永远为 `true` 而导致错误, 所以可选的只剩下返回一个对象了。

7.1.16 变量名为 618Software 是否合法

变量名 `618Software` 不合法。在 C 语言中, 变量名、函数名、数组名统称为标识符, C 语言规定标识符只能由字母 (`a~z`, `A~Z`)、数字 (`0~9`)、下画线 (`_`) 组成, 并且标识符的第一个字符必须是字母或下画线, 不能以数字开头, 不能包含除了 “`_`” 以外的任何特殊字符, 如 `%`、`#` 等, 不能包含空白字符 (换行符、空格和制表符)。

以下标识符都是非法的。

(1) `char`: `char` 是 C 语言的一个数据类型, 是保留字, 不能作为标识符, 其他的如 `int`、`float` 等类似。

(2) `number of book`: 标识符中不能有空格。

(3) `3com`: 以数字开头。

(4) `a*b`: `*` 不能作为标识符的字符。

值得注意的是, C 语言是区分大小写的, 例如 `Count` 与 `count` 被认为是两个不同的标识

符，这一点与其他语言不一样。

7.1.17 C 语言中，整型变量 x 小于 0，是否可知 x×2 也小于 0

假定计算机是 32 位的，用 2 的补码表示整数，若 $x < 0$ ，则 $x \times 2 < 0$ 不一定成立。例如，当 x 为整型值的最小时就不成立。

程序示例代码如下：

```
#include<stdio.h>
int main()
{
    int x = -4292967295;
    if (2*x<0)
        printf("2*x<0\n");
    else
        printf("2*x>0\n");
    return 0;
}
```

程序输出为
2*x>0

7.1.18 exit(status)是否跟从 main()函数返回的 status 等价

在 C 语言标准中，它们是等价的，但是如果在退出的时候需要使用 main()函数的局部数据，那么从 main()函数中使用 return()就不行了。

exit()函数与 return()的功能见表 7-1。

表 7-1 exit 函数与 return 的功能

| 函 数 | 功 能 |
|--------|---|
| return | 返回函数调用，如果返回的是 main()函数，则为退出程序 |
| exit | 在调用处强行退出程序，运行一次程序就结束。exit(0)是程序结束时返回 0 给系统，正常退出；exit(1)程序结束时返回 1 给系统；exit(n)程序结束时返回 n 给系统 |

对于 exit()函数而言，无论参数是几，其效果都是相同的，不同之处在于程序员可以用不同的数字来区别退出的原因，从而方便判断程序的异常问题。例如，内存分配失败是 exit(1)，打开文件失败是 exit(2)或者用来标示在此处退出。

7.1.19 已知 String 类定义，如何实现其函数体

String 类定义如下：

```
class String
{
public:
    String(const char *str = NULL);           //通用构造函数
    String(const String &another);           //复制构造函数
    ~String( );                               //析构函数
    String& operator =(const String &rhs);   //赋值函数
private:
    char *m_data;                             //用于保存字符串
};
```

在这个类中包括了指针类成员变量 m_data，所以需要自定义其复制构造函数、赋值运算操作符函数，避免单纯的指针值的复制。

具体而言，String 类的函数体实现代码如下：

```
#include <iostream>
using namespace std;

class String
{
public:
    String(const char *str = NULL);           //通用构造函数
    String(const String &another);           //复制构造函数
    ~String( );                             //析构函数
    String& operator =(const String &rhs);    //赋值函数
private:
    char *m_data;                           //用于保存字符串
};

String::String(const char* str)
{
    if(str == NULL)
    {
        m_data = new char[1];
        m_data[0] = '\0';
    }
    else
    {
        m_data = new char[strlen(str)+1];
        strcpy(m_data,str);
    }
}

String::String(const String &another)
{
    m_data = new char[strlen(another.m_data)+1];
    strcpy(m_data,another.m_data);
}

String::~~String( )
{
    delete[] m_data;
}

String& String::operator =(const String &rhs)
{
    if(this == &rhs)
        return *this;
    delete []m_data;
    m_data = new char[strlen(rhs.m_data)+1];
    strcpy(m_data,rhs.m_data);
    return *this;
}

int main( )
{
    String a("abcdefg");
    printf("%s\n",a);
    String b(a);
    printf("%s\n",b);
    String c=b;
```



```

        printf("%s\n",c);
        return 0;
    }

```

程序输出为

```

abcdefg
abcdefg
abcdefg

```

7.1.20 在 C++ 中如何实现模板函数的外部调用

`export` 是 C++ 新增的关键字，它的作用是实现模板函数的外部调用，类似于 `extern` 关键字。为了访问其他代码文件中的变量或对象，对普通类型（包括基本数据类、结构和类）可以利用关键字 `extern` 来使用这些变量或对象，但对于模板类型，则可以在头文件中声明模板类和模板函数，在代码文件中使用关键字 `export` 来定义具体的模板类对象和模板函数，然后在其他用户代码文件中，包含声明头文件后，就可以使用这些对象和函数了。使用方式如下：

```

extern int n;
extern struct Point p;
extern class A a;
export template <class T> class Stack<int> s;
export template<class T> void f(T&t){...}

```

7.1.21 在 C++ 中，关键字 `explicit` 有什么作用

在 C++ 中，如下声明是合法的。

```

class String
{
    String(const char* p);
    ...
}
String s1 = "hello";

```

上例中，`String s1 = "hello"` 会执行隐式转换，等价于 `String s1 = String("hello")`。为了避免这种情况的发生，C++ 引入了关键字 `explicit`，它可以阻止不应该允许的经过转换构造函数进行的隐式转换的发生，声明为 `explicit` 的构造函数不能在隐式转换中使用。

在 C++ 中，一个参数的构造函数（或者除了第一个参数外其余参数都有默认值的多参构造函数）一般具备两个功能：构造器和默认且隐含的类型转换操作符。所以，当 `AAA = XXX`，恰好 `XXX` 的类型正好是 `AAA` 单参数构造器的参数类型，这时候编译器就自动调用这个构造器，创建一个 `AAA` 的对象。而在某些情况下，却违背了程序员的本意。此时就要在这个构造器前面加上 `explicit` 修饰，指定这个构造器只能被明确地调用、使用，不能作为类型转换操作符被隐含地使用。

程序代码如下：

```

class Test1
{
    public:
        Test1(int n) { num = n; } //普通构造函数
    private:
        int num;
};

class Test2

```

```

{
    public:
        explicit Test2(int n) { num = n; }    //explicit(显式)构造函数
    private:
        int num;
};

int main( )
{
    Test1 t1 = 12;                          //隐式调用其构造函数, 成功
    Test2 t2 = 12;                          //编译错误, 不能隐式调用其构造函数
    Test2 t3(12);                          //显示调用成功
    return 0;
}

```

Test1 的构造函数带一个 int 型的参数, Test1 t1 = 12 会隐式转换成调用 Test1 的这个构造函数, 而 Test2 的构造函数被声明为 explicit (显式), 这表示不能通过隐式转换来调用这个构造函数, 因此 Test2 t2 = 12 会出现编译错误。普通构造函数能够被隐式调用, 而 explicit() 构造函数只能被显式调用。

7.1.22 C++中异常的处理方法以及使用了哪些关键字

C++异常处理使用的关键字有: try、catch、throw。C++中的异常处理机制只能处理由 throw 捕获的异常, 没有捕获的将被忽略。使用 try{}catch(){} 语句来捕获异常, 把可能发生异常的代码放在 try{} 语句块中, 后面跟若干个 catch(){} 负责处理具体的异常类型, 这样一组有 try 块和不少于一个的 catch 块就构成了一级异常捕获。如果本级没有带适当类型参数的 catch 块, 将不能捕获异常, 异常就会向上一级传递, 函数调用处如果没有捕获住异常, 则直接跳到更高一层的调用者, 如果一直没有捕获该异常, C++会使用默认的异常处理函数, 该函数可能会让程序最终跳出 main() 函数并导致程序异常终止。

catch 的作用是捕获异常, finally 不管代码是否有异常都执行。try 中如果有 return, 仍然需要执行 finally 语句。此种情况的执行过程如下:

- (1) 执行 return 返回语句 (return 之后的语句内容), 计算返回值, 暂存在一个临时变量中。
- (2) 执行 finally 语句块。
- (3) return 原来已经计算得到的结果值。

如果在 finally 区段中又调用了一次 return 语句, 则 try 区段中的返回值将会被遮掩, 使得方法调用者得到的是 finally 区段中的返回值, 这常常又与程序编写的初衷相背。

7.1.23 如何定义和实现一个类的成员函数为回调函数

回调函数就是被调用者回头调用的函数, 它是一个通过函数指针调用的函数。如果把函数的指针 (地址) 作为参数传递给另一个函数, 当这个指针被用为调用它所指向的函数时, 此时就可以称它为回调函数。回调函数不是由该函数的实现方直接调用的, 而是在特定的事件或条件发生时由另外的一方调用的, 用于对该事件或条件进行响应。

使用回调函数实际上就是在调用某个函数 (通常是 API 函数) 时, 将自己的一个函数 (这个函数为回调函数) 的地址作为参数传递给那个被调用函数。而该被调用函数在需要的时候, 利用传递的地址调用回调函数。

回调函数由程序员自己编写, 当需要调用另外一个函数时, 这个函数的其中一个参数就是

这个回调函数名。系统在必要的时候就会调用程序员写的回调函数，这样就可以在回调函数里完成要做的事。

要定义和实现一个类的成员函数为回调函数需要做 3 件事：

- (1) 声明。
- (2) 定义。
- (3) 设置触发条件，就是在函数中把回调函数名作为一个参数，以便系统调用。

声明回调函数类型示例如下：

```
typedef void (*FunPtr)(void);
//定义回调函数
class A
{
    public:
        //回调函数，必须声明为 static
        static void callBackFun(void)
        {
            ...
        }
};
//设置触发条件
void Funtype(FunPtr p)
{
    p();
}
void main(void)
{
    Funtype(A::callBackFun);
}
```

回调函数与应用程序接口（API）非常接近，它们都是跨层调用的函数，但区别是 API 是低层提供给高层的调用，一般这个函数对高层都是已知的。而回调函数正好相反，它是高层提供给底层的调用，对于低层它是未知的，必须由高层进行安装，这个安装函数其实就是一个低层提供的 API，安装后低层不知道这个回调的名字，但它通过一个函数指针来保存这个回调函数，在需要调用时，只需引用这个函数指针和相关的参数指针。

7.2 内存分配

系统蓝屏，很大原因都是系统自身代码有缺陷引起的，而系统代码缺陷很大程度上与内存分配不当有关。由于内存分配不当引起的堆栈溢出、缓冲区溢出等问题，常常会导致系统瘫痪甚至崩溃，所以理解内存分配对于一名合格的程序员而言非常有必要。

7.2.1 内存分配的形式有哪些

一个 C/C++ 编译的程序所占用的系统内存一般分为以下几个部分的内容：

(1) 由符号起始的区块（Block Started by Symbol, BSS）段：BSS 段通常是指用来存放程序中未初始化的全局数据和静态数据的一块内存区域。BSS 段属于静态内存分配，程序结束后静态变量资源由系统自动释放。

(2) 数据段（data segment）：数据段通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段也属于静态内存分配。

(3) 代码段（code segment/text segment）：代码段有时候也叫文本段，通常是指用来存放

程序执行代码（包括类成员函数和全局函数以及其他函数代码）的一块内存区域，这部分区域的大小在程序运行前就已经确定，并且内存区域通常是只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，如字符串常量。这个段一般是可以被共享的，如在 Linux 系统中打开了两个 Vi 来编辑文本，那么一般来说这两个 Vi 是共享一个代码段的。

（4）堆（heap）：堆用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用 malloc 或 new 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张），当利用 free 或 delete 等函数释放内存时，被释放的内存从堆中被删除（堆被缩减）。堆一般由程序员分配释放，若程序员不释放，程序结束时可能由操作系统回收。需要注意的是，它与数据结构中的堆是两回事，分配方式类似于链表。

（5）栈（stack）：栈用户存放程序临时创建的局部变量，一般包括函数括弧“{}”中定义的变量（但不包括 static 声明的变量，static 意味着在数据段中存放变量）。除此之外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且等到调用结束后，函数的返回值也会被存放回栈中。栈由编译器自动分配释放，存放函数的参数值、局部变量的值等。其操作方式类似于数据结构中的栈。栈内存分配运算内置于处理器的指令集中，一般使用寄存器来存取，效率很高，但是分配的内存容量有限。

需要注意的是，代码段和数据段之间有明确的分隔，但是数据段和堆栈段之间没有，而且栈是向下增长的，堆是向上增长的。程序示例如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int global=0;           //全局初始化区
char *p1;               //全局未初始化区
int main()
{
    int a;               //栈
    char s[]="abcdefg";  //栈
    char *p2;            //栈
    char *p3="123456789"; //123456789 在常量区，p3 在栈上
    static int c=0;       //全局（静态）初始化区
    p1=(char *)malloc(100);
    p2=(char *)malloc(200); //分配得来的 100 和 200 字节的区域就在堆区
    strcpy(p1,"123456789"); //123456789 放在常量区，编译器可能会将它与 p3 所指向的
                           // "123456789"优化成一个地方
    return 0;
}
```

除了全局静态对象，还有局部静态对象和类的静态成员，局部静态对象是在函数中定义的，就像栈对象一样，只不过，其前面多了个 static 关键字。局部静态对象的生命期是从其所在函数第一次被调用，更确切地说，是当第一次执行到该静态对象的声明代码时，产生该静态局部对象，直到整个程序结束时，才销毁该对象。类的静态成员的生命周期是该类的第一次调用到程序的结束。

7.2.2 什么是内存泄露

堆是动态分配内存的，并且可以分配使用很大的内存，使用不好会产生内存泄露。频繁地使用 malloc 和 free 会产生内存碎片（类似磁盘碎片）。

所谓内存泄露 (memory leak) 是指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。一般常说的内存泄露是指堆内存的泄露, 内存泄露其实并非指内存存在物理上的消失, 而是应用程序分配某段内存后, 由于设计错误, 失去了对该段内存的控制, 因而造成了内存的浪费。内存泄露与许多其他问题有着相似的症状, 并且通常情况下只能由那些可以获得程序源代码的程序员才可以分析出来。

应用程序一般使用 `malloc`、`calloc`、`realloc`、`new` 等函数从堆中分配到一块内存, 使用完后, 程序必须负责相应地调用 `free` 或 `delete` 释放该内存块, 否则这块内存就不能被再次使用, 造成内存泄露。

例如, 对指针进行重新赋值, 程序代码如下:

```
char *memoryArea = malloc(10);
char *newArea = malloc(10);
memoryArea = newArea;
```

对 `memoryArea` 的赋值会导致 `memoryArea` 之前指向的内容丢失, 最终造成内存泄露。如下程序就因为未能对返回值进行处理, 最终导致内存泄露。

```
char *fun()
{
    return malloc(20);
}
void callfun()
{
    fun();
}
```

内存泄露往往会导致系统出现 CPU 资源耗尽的严重后果, 所以开发人员在编码过程中要养成良好的编程习惯, 用 `malloc` 或 `new` 分配的内存都应当在适当的时机用 `free` 或 `delete` 释放, 在对指针赋值前, 要确保没有内存位置会变为孤立的。每当释放结构化的元素, 而该元素又包含指向动态分配的内存位置的指针时, 都应首先遍历子内存位置并从那里开始释放, 然后再遍历回父节点, 始终正确处理返回动态分配的内存引用的函数返回值。

7.2.3 栈空间的最大值是多少

在 Windows 下, 栈是向低地址扩展的数据结构, 是一块连续的内存的区域。栈顶的地址和栈的最大容量是系统预先规定好的, 在 Windows 下, 栈的大小是 2MB。而申请堆空间的大小一般小于 2GB。

由于内存的读取速度比硬盘快, 当程序遇到大规模数据的频繁存取时, 开辟内存空间很有作用。栈的速度快, 但是空间小, 不灵活。堆是向高地址扩展的数据结构, 是不连续的内存区域。这是由于系统是用链表来存储空闲内存地址的, 自然是不连续的, 而链表的遍历方向是由低地址向高地址的, 而堆的大小受限于计算机系统中有用的虚拟内存, 所以堆获得的空间比较灵活, 也比较大, 但是速度相对慢一些。VC 中堆是人为控制的, 所以容易产生内存泄露的问题。

一般情况下, 可以通过以下两种方法更改栈的大小。

- (1) link 时用 `/STACK` 指定它的大小, 或者在 `.def` 中使用 `STACKSIZE` 指定它的大小。
- (2) 使用控制台命令 “`EDITBIN`” 更改 exe 的栈空间大小。

需要注意的是, Linux 默认栈空间大小为 8MB, 通过命令 `ulimit -s` 来设置。

7.2.4 什么是缓冲区溢出

缓冲区是程序运行的时候机器内存中的一个连续块, 它保存了给定类型的数据, 随着动态

分配变量会出现问题。缓冲区溢出是指当向缓冲区内填充数据位数超过了缓冲区自身的容量限制时，发生的溢出的数据覆盖在合法数据（数据、下一条指令的指针、函数返回地址等）上的情况。最好的情况是程序不允许输入超过缓冲区长度的字符并检查数据长度，由于大多数程序都会假设数据长度总是与所分配的储存空间相当，进而存在缓冲区溢出安全隐患。

程序示例如下：

```
#include <unistd.h>
void Test( )
{
    char buff[4];
    printf("Some input: ");
    gets(buff);
    puts(buff);
}

int main(int argc, char *argv[ ])
{
    Test( );
    return 0;
}
```

该程序的 Test() 函数中使用了标准的 C 语言输入函数 gets()，由于它没有执行边界检查，最终会导致 Test() 函数存在缓冲区溢出安全漏洞。Test() 函数的缓冲区最多只能容纳 3 个字符和一个空字符，所以超过 4 个字符就会造成缓冲区溢出。该程序的堆栈说明如图 7-1 所示。buff[4] 被存储在 Test() 函数的栈帧里。

如果输入 3 个字符“AAA”，buffer[4] 正好被 3 个字符和 NULL 填充。输入 3 个字符后的堆栈说明，如图 7-2 所示。

如果输入 5 个字符“AAAA”，%ebp 的一些区域的内容将被覆盖，造成缓冲区溢出。输入 5 个字符后的堆栈说明，如图 7-3 所示。

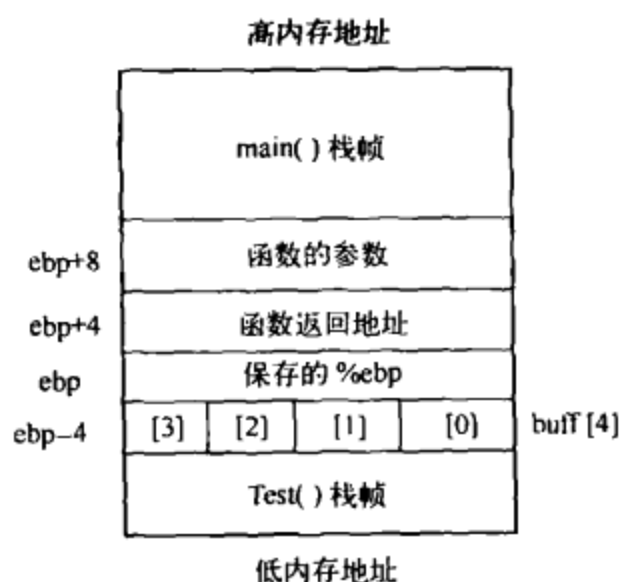


图 7-1 堆栈说明

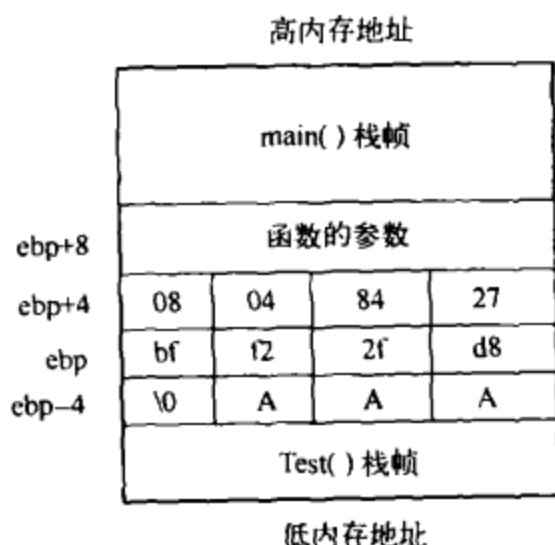


图 7-2 输入 3 个字符后的堆栈说明

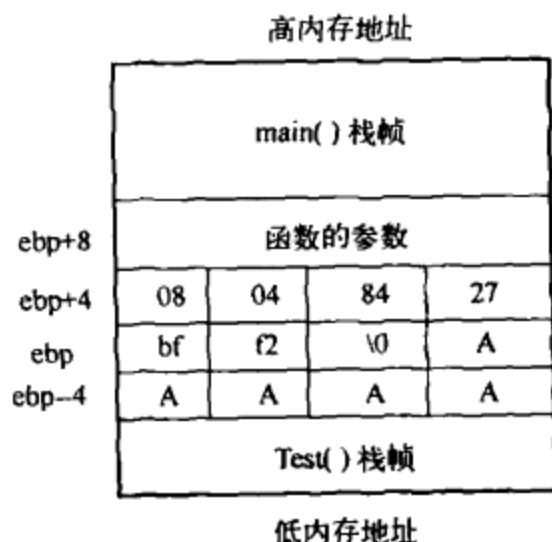


图 7-3 输入 5 个字符后的堆栈说明

人为的缓冲区溢出一般是由于攻击者写一个超过缓冲区长度的字符串植入到缓冲区，然后再向一个有限空间的缓冲区中植入超长的字符串，这时可能会出现两个结果：一是过长的字符串覆盖了相邻的存储单元，引起程序运行失败，严重的可导致系统崩溃；另一个结果就是利用这种漏洞可以执行任意指令，甚至可以取得系统 root 特级权限，进而危害系统安全。

缓冲区溢出是目前导致“黑客”型病毒横行的主要原因。红色代码、Slammer、“冲击波”都是利用缓冲区溢出漏洞的典型。防止利用缓冲区溢出发起的攻击，关键在于程序开发者在开发程序时仔细检查溢出情况，不允许数据溢出缓冲区。

7.3 sizeof

有时候喜欢称呼胖人为胖子，该称呼并非恶意，只是一种亲昵的叫法而已，其实胖人除了体积较常人大一些以外，其生活与常人也不会有大的区别。但对于变量而言，sizeof 的大小就像变量的体积一样，它的大小直接影响着变量的存储与访问效率。

7.3.1 sizeof 是关键字吗

在回答该问题前，首先弄清楚一个问题，什么是关键字？C 语言的所有命令、系统函数名等被称为 C 语言的关键字。C 语言一共有 32 个关键字，5 种语言类型，具体内容见表 7-2。

表 7-2 C 语言关键字

| 类 型 | 关 键 字 |
|------------|---|
| 数据类型 14 个 | void, char, int, float, double, short, long, signed, unsigned, struct, union, enum, typedef, sizeof |
| 控制类别 6 个 | auto, static, extern, register, const, volatile |
| 控制关键字 12 个 | return, continue, break, goto, if, else, switch, case, default, for, do, while |

根据上表可知，sizeof 是数据类型的关键字，而非函数，很多时候它都可能被误解是操作符，这是不对的。

引申：预处理指令是否是 C 语言中的语言类型？

不是。语句是编程语言的基础，C 语言中的语言类型一共有以下 5 种：

- (1) 表达式语句。
 - (2) 函数调用语句。
 - (3) 控制语句。if 语句、switch 语句（这两种为条件判断语句）、do while 语句、while 语句、for 语句（这 3 种为循环执行语句）、break 语句、continue 语句、return 语句、goto 语句（这 4 种为转向语句）。
 - (4) 复合语句。
 - (5) 空语句。
- 需要注意的是，由于预处理指令的结尾不能添加分号，所以预处理指令不是语句。

7.3.2 strlen("\0")=? sizeof("\0")=?

strlen("\0") = 0, sizeof("\0") = 2。

strlen 执行的是一个计数器的工作，它从内存的某个位置（可以是字符串开头，中间某个位置，甚至是某个不确定的内存区域）开始扫描，直到碰到第一个字符串结束符'\0'为止，然后返回计数器值。

sizeof 是 C 语言的关键字，它以字节的形式给出了其操作数的存储大小，操作数可以是一个表达式或括在括号内的类型名，操作数的存储大小由操作数的类型决定。

具体而言, `strlen` 与 `sizeof` 的差别表现在以下几个方面:

(1) `sizeof` 是关键字, 而 `strlen` 是函数。 `sizeof` 后如果是类型必须加括弧, 如果是变量名可以不加括弧。

(2) `sizeof` 操作符的结果类型是 `size_t`, 它在头文件中 `typedef` 为 `unsigned int` 类型。该类型保证能够容纳实现所建立的最大对象的字节大小。

(3) `sizeof` 可以用类型作为参数, `strlen` 只能用 `char*` 做参数, 而且必须是以 `"\0"` 结尾的。 `sizeof` 还可以以函数作为参数, 如 `intg()`, 则 `sizeof(g())` 的值等于 `sizeof(int)` 的值, 在 32 位计算机下, 该值为 4。

(4) 当数组名做 `sizeof` 的参数时不退化, 传递给 `strlen` 就退化为指针了。以数组 `char a[10]` 为例, 在 32 位机器下, `sizeof(a)=1*10=10`, 而传递给 `strlen` 就不一样了。

(5) 大部分编译程序的 `sizeof` 都是在编译的时候计算的, 所以可以通过 `sizeof(x)` 来定义数组维数。而 `strlen` 的计算则是在运行期计算的, 用来计算字符串的实际长度, 不是类型占内存的大小。例如, `char str[20]="0123456789"`, 字符数组 `str` 是编译期大小已经固定的数组, 在 32 位机器下, 为 `1*20=20`, 而其 `strlen` 大小则是在运行期确定的, 所以其值为字符串的实际长度 10。

(6) 当用于计算一个结构类型或变量的 `sizeof` 时, 返回实际的大小, 当用于静态数组时, 返回数组的维数, 而 `sizeof` 不能返回动态数组大小。

(7) 数组作为参数传给函数时传的是指针而不是数组, 传递的是数组的首地址。例如:

```
fun(char [8])
fun(char [])
```

都等价于 `fun(char *)`。在 C++ 里参数传递数组永远都是传递指向数组首元素的指针, 编译器不知道数组的大小, 如果想在函数内知道数组的大小, 需要这样做: 进入函数后用 `memcpy` 复制出来, 长度由另一个形参传进去。

```
fun(unsigned char *p1, int len)
{
    unsigned char* buf = new unsigned char[len+1];
    memcpy(buf, p1, len);
}
```

程序示例 1:

```
#include <stdio.h>
#include <string.h>

int main()
{
    char arr[10] = "Hello";
    printf("%d\n", strlen(arr));
    printf("%d\n", sizeof(arr));
    return 0;
}
```

程序输出结果:

```
5
10
```

`sizeof` 返回定义的 `arr` 数组时, 编译器为其分配的数组空间大小不关心里面存了多少数据。 `strlen` 只关心存储的数据内容, 不关心空间的大小和类型。

程序示例 2:

```
#include <stdio.h>
#include <string.h>
```



```
int main( )
{
    char * parr = new char[10];
    printf("%d\n", strlen(parr));
    printf("%d\n", sizeof(parr));
    printf("%d\n", sizeof(*parr));
    return 0;
}
```

程序输出结果：

```
14
4
1
```

在上例中，程序定义了一个字符指针 `parr`，它指向一个分配了 10 个空间的字符数组，由于没有进行初始化，根据 `strlen` 的计算原理，所以不能够确定 `strlen(parr)` 的值，因为无法确定字符串的终止位置，所以该值为一个随机值，本例中输出为 14。在 32 位机器下，`parr` 为一个指针，所以 `sizeof(parr)` 的值为 4，`parr` 为指向字符的指针，所以 `sizeof(*parr)` 的值为 1。

7.3.3 对于结构体而言，为什么 `sizeof` 返回的值一般大于期望值

`struct` 是一种复合数据类型，其构成元素既可以是基本数据类型，如 `int`、`double`、`float`、`short`、`char` 等，也可以是复合数据类型，如数组、`struct`、`union` 等数据单元。

一般而言，`struct` 的 `sizeof` 是所有成员对齐后长度相加，而 `union` 的 `sizeof` 是取最大的成员长度。

在结构中，编译器为结构的每个成员按其自然边界（`alignment`）分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同。

字节对齐也称为字节填充，它是 C++ 编译器的一种技术手段，主要是为了在空间与复杂度上达到平衡。简单地讲，是为了在可接受的空间浪费的前提下，尽可能地提高对相同运算过程的最少（快）处理。字节对齐的作用不仅是便于 CPU 的快速访问，使 CPU 的性能达到最佳，同时合理地利用字节对齐可以有效地节省存储空间。例如，32 位的计算机的数据传输值是 4 字节，64 位计算机数据传输是 8 字节，这样 `struct` 在默认的情况上，编译器会对 `struct` 的结构进行（32 位机）4 的倍数或（64 位机）8 的倍数的数据对齐。对于 32 位机来说，4 字节对齐能够使 CPU 访问速度提高，比如说一个 `long` 类型的变量，如果跨越了 4 字节边界存储，那么 CPU 要读取两次，这样效率就低了，但需要注意的是，如果在 32 位机中使用 1 字节或者 2 字节对齐，不仅不会提高效率，反而会使变量访问速度降低。

在默认情况下，编译器为每一个变量或数据单元按其自然对界条件分配空间。一般地，可以通过下面的方法来改变默认的对界条件：

（1）使用伪指令 `#pragma pack (n)`，C 编译器将按照 `n` 个字节对齐。

（2）使用伪指令 `#pragma pack ()`，取消自定义字节对齐方式。

（3）另外，还有如下的一种方式：`_attribute((aligned (n)))`，让所作用的结构成员对齐在 `n` 字节自然边界上。如果结构中有成员的长度大于 `n`，则按照最大成员的长度来对齐。`_attribute__((packed))`，取消结构在编译过程中的优化对齐，按照实际占用字节数进行对齐。

例如如下数据结构：

```
struct test
{
    char x1;
```

```

short x2;
float x3;
char x4;
};

```

由于编译器默认情况下会对 `struct` 作边界对齐, 结构的第一个成员 `x1`, 其偏移地址为 0, 占据了第 1 个字节, 第二个成员 `x2` 为 `short` 类型, 其起始地址必须 2 字节对齐, 因此编译器在 `x2` 和 `x1` 之间填充了一个空字节。结构的第三个成员 `x3` 和第四个成员 `x4` 恰好落在其自然边界地址上, 在它们前面不需要额外的填充字节。在 `test` 结构中, 成员 `x3` 要求 4 字节对齐, 是该结构所有成员中要求的最大边界单元, 因而 `test` 结构的自然对齐条件为 4 字节, 编译器在成员 `x4` 后面填充了 3 个空字节。整个结构所占据空间为 12 字节。

再例如如下数据结构:

```

struct s1
{
    short d;
    int a;
    short b;
}a1;

```

在 32 位机器下, `short` 型占两个字节, `int` 型占 4 个字节, 所以为了满足字节对齐, 变量 `d` 除了自身所占用的两个字节外, 还需要再填充两个字节, 变量 `a` 占用 4 个字节, 变量 `b` 除了自身占用的两个字节, 还需要两个填充字节, 所以最终 `s1` 的 `sizeof` 为 12。

字节对齐的细节和编译器实现相关, 但一般而言, 满足以下 3 个准则:

- (1) 结构体变量的首地址能够被其最宽基本类型成员的大小所整除。
- (2) 结构体每个成员相对于结构体首地址的偏移量 (`offset`) 都是成员大小的整数倍, 如有需要编译器会在成员之间加上填充字节。
- (3) 结构体的总大小为结构体最宽基本类型成员大小的整数倍, 如有需要编译器会在最末一个成员之后加上填充字节。

需要注意的是, 基本类型是指前面提到的像 `char`、`short`、`int`、`float`、`double` 这样的内置数据类型, 这里所说的“数据宽度”就是指其 `sizeof` 的大小, 在 32 位机器上, 这些基本数据类型的 `sizeof` 大小分别为 1、2、4、4、8。由于结构体的成员可以是复合类型, 所以在寻找最宽基本类型成员时, 应当包括复合类型成员的子成员, 而不是把复合成员看成是一个整体。如果一个结构体中包含另外一个结构体成员, 那么此时最宽基本类型成员不是该结构体成员, 而是取基本类型的最宽值。但在确定复合类型成员的偏移位置时, 则是将复合类型作为整体看待, 即复杂类型 (如结构) 的默认对齐方式是它最长的成员的对齐方式, 这样在成员是复杂类型时, 可以最小化长度, 达到程序优化的目的。

7.3.4 指针进行强制类型转换后与地址进行加法运算, 结果是什么

假设在 32 位机器上, 在对齐为 4 的情况下, `sizeof(long)` 的结果为 4 字节, `sizeof(char *)` 的结果为 4 字节, `sizeof(short int)` 的结果与 `sizeof(short)` 的结果都为 2 字节, `sizeof(char)` 的结果为 1 字节, `sizeof(int)` 的结果为 4 字节, 由于 32 位机器上是 4 字节对齐, 以如下结构体为例:

```

struct BBB
{
    long num;
    char *name;
    short int data;
    char ha;
}

```

```
short ba[5];
}*p;
```

当 $p=0x1000000$; 则 $p+0x200=?$ $(Ulong)p+0x200=?$ $(char*)p+0x200=?$

其实, 在 32 位机器下, $sizeof(struct\ BBB)=sizeof(*p)=4+4+2+1+1/*\text{补齐}*/+2*5+2/*\text{补齐}*/=24$ 字节, 而 $p=0x1000000$, 那么 $p+0x200=0x1000000+0x200*24$ 指针加法, 加出来的是指针类型的字节长度的整倍数。就是 p 偏移 $sizeof(p) * 0x200$ 。

$(Ulong)p+0x200=0x1000000+0x200$ 经过 $Ulong$ 后, 这已经不再是指针加法, 而变成一个数值加法了。

$(char*)p+0x200=0x1000000+0x200*4$ 结果类型是 $char*$, 这儿的 4 是 $char*$ 数据类型, 占用 4 个字节。

7.4 指针

用指针变量可以表示各种数据结构, 能很方便地使用数组、字符串和链表, 并能像汇编语言一样处理内存地址, 从而编写出精练而高效的程序。但是由于指针并不是直接操作数据, 而且它可以直接与内存打交道, 使用稍有不慎, 就会造成严重的后果——程序崩溃, 所以在使用指针时一定要深刻理解与指针相关的一些问题。

7.4.1 使用指针有哪些好处

指针与其他类型变量一样, 不同之处在于一般的变量包含的是实际的真实数据, 而指针包含的是一个指向内存中某个位置的地址。指针好处众多, 一般而言, 使用指针有以下几个方面的的好处:

- (1) 可以动态分配内存。
- (2) 进行多个相似变量的一般访问。
- (3) 为动态数据结构, 尤其是树和链表, 提供支持。
- (4) 遍历数组, 如解析字符串。

(5) 高效地按引用“复制”数组与结构, 特别是作为函数参数的时候, 可以按照引用传递函数参数, 提高开发效率。

7.4.2 引用还是指针

程序设计中的引用其实就是别名的意思, 它用于定义一个变量来共享另一个变量的内存空间, 变量是一个内存空间的名字, 如果给内存空间起另外一个名字, 那就能够共享这个内存了, 进而提高程序的开发效率。指针指向另一个内存空间的变量, 可以通过它来索引另一个内存空间的内容, 而指针本身也有自己的内存空间。

引用与指针有着相同的地方, 即指针指向一块内存, 它的内容是所指内存的地址, 引用是某块内存的别名。但是, 两者并非完全相同, 它们之间也存在着差别, 具体表现在以下几个方面:

- (1) 从本质上讲, 指针是存放变量地址的一个变量, 在逻辑上是独立的, 它可以被改变, 即其所指向的地址可以被改变, 其指向的地址中所存放的数据也可以被改变。而引用则只是一个别名而已, 它在逻辑上不是独立的, 它的存在具有依附性, 所以引用必须在一开始就被初始化, 而且其引用的对象在其整个生命周期中是不能被改变的, 即自始至终只能依附于同一个变量, 具有“从一而终”的特性。

(2) 作为参数传递时, 两者不同。在 C++ 语言中, 指针与引用都可以用于函数的参数传递, 但是指针传递参数和引用传递参数有着本质的不同。指针传递参数本质上是值传递的方式, 它所传递的是一个地址值。值传递过程中, 被调函数的形式参数作为被调函数的局部变量处理, 即在栈中开辟了内存空间以存放由主调函数放进来的实参的值, 从而成为了实参的一个副本。值传递的特点是被调函数对形式参数的任何操作都是作为局部变量进行的, 不会影响主调函数的实参变量的值。而在引用传递过程中, 被调函数的形式参数虽然也作为局部变量在栈中开辟了内存空间, 但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址, 即通过栈中存放的地址访问主调函数中的实参变量。正因为如此, 被调函数对形参做的任何操作都影响了主调函数中的实参变量。虽然它们都是在被调函数栈空间上的一个局部变量, 但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。而对于指针传递的参数, 如果改变被调函数中的指针地址, 它将影响不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量, 那就得使用指向指针的指针, 或者指针引用。

(3) 引用使用时不需要解引用 (*), 而指针需要解引用。

(4) 引用只能在定义时被初始化一次, 之后不能被改变, 即引用具有“从一而终”的特性。而指针却是可变的, 指针的初始化不是指指针的定义, 而是指针变量存储的数值是个无效的数值。例如, 定义 `float a`, 该语句表示 `a` 会分配一个地址, 但初始值是一个随机的值, 同样, `float *a` 也会为 `a` 分配一个地址, 初始值也是随机的值, 初始化可以将 `a = NULL`, 这样在以后的程序中可以增加 `if(a == NULL)` 来判断指针是否有效, 否则不行, 或者为指针分配或者指定空间, 如 `float *a = new float` 或者 `float b; float *a = &b`, 都可以为指针指向一块内存以实现初始化。

(5) 引用不可以为空, 而指针可以为空。引用必须与存储单元相对应, 一个引用对应一个存储单元。

(6) 对引用进行 `sizeof` 操作得到的是所指向的变量 (对象) 的大小, 而对指针进行 `sizeof` 操作得到的是指针本身 (所指向的变量或对象的地址) 的大小, `typeid(T) == typeid(T&)` 恒为真, `sizeof(T) == sizeof(T&)` 恒为真, 但是当引用作为成员时, 其占用空间与指针相同。

(7) 指针和引用的自增 (++) 运算意义不一样。

(8) 如果返回动态分配的对象或内存, 必须使用指针, 引用可能引起内存泄露。

由于引用与指针的区别, 所以并非所有使用指针的地方都可以使用引用, 也并非所有使用引用的地方都可以使用指针, 两者的使用也有其特定的环境。以如下实例为例进行分析。

```
(1) int *a; int * &p=a; int b=8; p=&b; //正确, 指针变量的引用
    void &a=3; //不正确, 没有变量或对象的类型是 void
    int &ri=NULL; //不正确, 有空指针, 无空引用

(2) int &ra=int; //不正确, 不能用类型来初始化
    int *p=new int; int &r=*p; //正确

(3) 引用不同于一般变量, 下面的类型声明是非法的:
    int &b[3]; //不能建立引用数组
    int &*p; //不能建立指向引用的指针
    int &&r; //不能建立引用的引用
```

(4) 当使用 `&` 运算符取一个引用的地址时, 其值为所引用变量的地址。

通过上面的实例可以发现, 引用与指针都有其特定的使用场景, 所以该使用指针时就使用指针, 该使用引用时就使用引用, 切不可混淆之。

7.4.3 指针和数组是否表示同一概念

指针可以随时指向任意类型的内存块，而数组可以在静态存储区被创建。例如，全局数组可以在栈上被创建。从原理与定义上看，虽然指针与数组表示的是不同的概念，但指针却可以方便地访问数组或者模拟数组，两者存在着一种貌似等价的关系，但也存在着诸多不同之处，主要表现在以下两个方面：

(1) 修改内容不同。

例如，`char a[] = "hello"`，可以通过取下标的方式对其元素值进行修改。例如，`a[0] = 'X'`是正确的，而对于 `char *p = "world"`，此时 `p` 指向常量字符串，所以 `p[0] = 'X'`是不允许的，编译会报错。

(2) 所占字节数不同。

例如，`char *p = "world"`，`p` 为指针，则 `sizeof(p)`得到的是一个指针变量的字节数，而不是 `p` 所指的内存容量。C/C++语言没有办法知道指针所指的内存容量，除非在申请内存时标记出来。

```
char a[] = "hello world";
char *p = a;
```

在 32 位机器上，`sizeof(a)=12` 字节，而 `sizeof(p)=4` 字节。

但需要注意的是，当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。

```
void Func(char a[100])
{
    cout<< sizeof(a);
}
```

此时 `sizeof(a)=sizeof(int)=4`，而不是 `sizeof(int)*100=400`。

7.4.4 指针是否可进行>、<、>=、<=、==运算

不可以。对于指针而言，只能进行==和!=运算。

7.4.5 指针与数字相加的结果是什么

为了更有说服力地解释本题结果，本题首先将题目程序完善，在 VC++6.0 的环境下编译运行，程序源代码如下：

```
#include <stdio.h>
int main( )
{
    unsigned char *p1;
    unsigned long *p2;
    p1=(unsigned char *)0x801000;
    p2=(unsigned long *)0x810000;
    printf("%x\n",p1+5);
    printf("%x\n",p2+5);
    return 0;
}
```

程序输出结果如下：

```
801005
810014
```

`p1=(unsigned char *)0x801000`，是给指针变量赋值，即把十六进制 `0x801000` 放到字符指针变量中，即指针变量 `p1` 的值就是 `0x801000`。

`p2=(unsigned long*)0x810000`，也是给指针变量赋值，同上。

输出结果 `p1+5` 的值是 801005，因为指针变量指向的值字符加 1 表示指针向后移动 1 个字节，那么加 5 代表向后移动 5 个字节，所以输出 801005。

`p2+5` 的值是 801016，因为指针变量指向的是长整型，加 1 表示指针向后移动 4 个字节，那么加 5 代表向后移动 $5 \times 4 = 20$ 个字节，所以输入 810014（十六进制）。

需要注意的是，内存的基本单位是字节，它以字节为存储单位储存，每个字节是 8 个二进制位，即 8 个 bit。

7.4.6 野指针？空指针

野指针是指指向不可用内存的指针。任何指针变量在被创建时，不会自动成为 NULL 指针（空指针），其默认值是随机的，所以指针变量在创建的同时应当被初始化，或者将指针设置为 NULL，或者让它指向合法的内存，而不应该放之不理，否则就会成为野指针。而同时由于指针被释放（`free` 或 `delete`）后，未能将其设置为 NULL，也会导致该指针变为野指针。虽然 `free` 和 `delete` 把指针所指的内存给释放掉了，但它们并没有把指针本身释放掉，一般可以采用语句 `if (p != NULL)` 进行防错处理，但是 `if` 语句却起不到防错作用，因为即使 `p` 不是 NULL 指针，它也不指向合法的内存块。第三种造成野指针的原因是指针操作超越了变量的作用范围。

程序示例如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( )
{
    char *p = (char *) malloc(100);
    strcpy(p, "hello");
    free(p);
    if(p != NULL)
        printf("Not NULL\n");
    return 0;
}
```

程序输出：

Not NULL

上例中，虽然对 `p` 执行了 `free` 操作，`p` 所指的内存被释放掉了，但是 `p` 所指的地址仍然不变，在后续的判断 `p` 是否为 NULL 时，根本没有起到防错的作用，所以程序输出仍然为 Not NULL。

空指针是一个特殊的指针，也是唯一一个对任何指针类型都合法的指针。指针变量具有空指针值，表示它当时处于闲置状态，没有指向有意义的内容。为了提高程序的可读性，标准库定义了一个与 0 等价的符号常量 NULL，程序里可以写 `p = 0` 或者 `p = NULL`，两种写法都把 `p` 置为空指针值。C 语言保证这个值不会是什么对象的地址。给指针值赋零则使它不再指向任何有意义的东西。

作为一种风格，很多程序员一般不愿意在程序中到处出现未加修饰的 0，所以习惯定义预处理宏 NULL（在 `<stdio.h>` 和其他几个头文件中）为空指针常数，通常是 0 或者 `((void *)0)`。希望区别整数 0 和空指针 0 的程序员可以在需要空指针的地方使用 NULL。

通用指针可以指向任何类型的变量。通用指针的类型用 `(void *)` 表示，因此也称为 void 指针。

程序代码如下：

```
#include <stdio.h>

int main( )
{
    int n=3, *p;
    void *gp;
    gp = &n;
    p=(int *)gp;
    printf("%d\n",*p);
    return 0;
}
```

程序输出结果:

3

7.5 预处理

预处理也称为预编译，它为编译做预备工作，主要进行代码文本的替换工作，用于处理#开头的指令，其中预处理器产生编译器的输出。表 7-3 所示为常见的一些预处理指令及其功能。

表 7-3 指令功能表

| 指 令 | 功 能 |
|----------|---|
| # | 空指令，无任何效果 |
| #include | 包含一个源代码文件，把源文件中的#include 扩展为文件正文，即把包含的.h 文件找到并扩展到#include 所在处 |
| #define | 定义宏 |
| #undef | 取消已定义的宏 |
| #if | 条件编译指令，如果给定条件为真，则编译下面代码 |
| #ifdef | 条件编译指令，如果宏已经定义，则编译下面代码 |
| #ifndef | 条件编译指令，如果宏没有定义，则编译下面代码 |
| #elif | 条件编译指令，如果前面的#if 给定条件不为真，当前条件为真，则编译下面代码 |
| #endif | 结束一个#if...#else 条件编译块 |
| #error | 停止编译并显示错误信息 |

经过预处理器处理的源程序与之前的源程序会有所不同，在预处理阶段所进行的工作只是纯粹地替换与展开，没有任何计算功能，所以在学习#define 命令时只有真正地理解这一点，才不会对此命令引起误解并误用。

7.5.1 C/C++头文件中的 ifndef/define/endif 的作用有哪些

如果一个项目中存在两个 C 文件，而这两个 C 文件都 include（包含）了同一个头文件，当编译时，这两个 C 文件要一同编译成一个可运行文件，可能会产生大量的声明冲突。而解决的办法是把头文件的内容都放在#ifndef 和#endif 中，一般格式如下：

```
#ifndef <标识>
#define <标识>
...
...
#endif
```

上述代码的作用是当“当标识没有由#define 定义过时，则定义标识。<标识>在理论上来

说可以是自由命名的，但每个头文件的这个“标识”都应该是唯一的。标识的命名规则一般是头文件名全大写，前后加下画线，并把文件名中的“.”也变成下画线，如 `stdio.h`。

```
#ifndef _STDIO_H_
#define _STDIO_H_
...
#endif
```

在 `#ifndef` 中定义变量出现的问题（一般不定义在 `#ifndef` 中）如下所示。

```
#ifndef AAA
#define AAA
...
int i;
...
#endif
```

里面有一个变量定义，在 VC 中链接时就出现了 `i` 重复定义的错误，而在 C 语言中成功编译。

7.5.2 `#include <filename.h>` 和 `#include "filename.h"` 有什么区别

对于 `#include <filename.h>`，编译器先从标准库路径开始搜索 `filename.h`，然后从本地目录搜索，使得系统文件调用较快。而对于 `#include "filename.h"`，编译器先从用户的工作路径开始搜索 `filename.h`，后去寻找系统路径，使得自定义文件较快。

引申：头文件的作用有哪些？

头文件的作用主要表现为以下两个方面：

(1) 通过头文件来调用库功能。出于对源代码保密的考虑，源代码不便（或不准）向用户公布，只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能，而不必关心接口是怎么实现的。编译器会从库中提取相应的代码。

(2) 头文件能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，能大大减轻程序员调试、改错的负担。

7.5.3 `#define` 有哪些缺陷

由于宏定义在预处理阶段进行，主要做的是字符替换工作，所以它存在着一些固有的缺陷：

(1) 它无法进行类型检查。宏定义是在编译前进行字符的替换，因为还没编译，不能编译前就检查好类型是否匹配，而只能在编译时才知道，所以不具备类型检查功能。

(2) 由于优先级的不同，使用宏定义时，可能会存在副作用。例如，执行加法操作的宏定义运算 `#define ADD(a,b) a+b` 在使用的过程中，对于表达式的运算就可能存在潜在的问题，而应该改为 `#define ADD(a,b) ((a)+(b))`。

(3) 无法单步调试。

(4) 会导致代码膨胀。由于宏定义是文本替换，需要对代码进行展开，相比较函数调用的方式，会存在较多的冗余代码。

(5) 在 C++ 中，使用宏无法操作类的私有数据成员。

7.5.4 如何使用 `define` 声明一个常数，用以表明 1 年中有多少秒（忽略闰年问题）

```
#define SECOND_PER_YEAR (60 * 60 * 24 * 365)UL
```


在以上定义中，需要注意以下 3 个方面的内容：

(1) 由于宏定义是预处理指令，而非语句，所以在进行宏定义时，不能以分号结束。

(2) 预处理只会执行简单的替换，不会计算表达式的值，所以需要注意括号的使用，直接写出是如何计算出一年中有多少秒而不是计算出实际的值。

例如：

```
#define N 4+5
cout<<2*N;
```

如果预处理计算表达式的值，那么输出结果应该是 $2 \times (4+5)$ ，等于 18，可是实际输出结果却是 $2 \times 4+5$ ，等于 13。

(3) 考虑到可能存在数据溢出问题，更加规范化的写法是使用长整型类型，即 UL 类型，告诉编译器这个常数是长整型数。

7.5.5 含参数的宏与函数有什么区别

含参数的宏有时完成的是函数实现的功能，但是并非所有的函数都可以被含参数的宏所替代。具体而言，含参数的宏与函数的特点如下：

(1) 函数调用时，首先求出实参表达式的值，然后带入形参。而使用带参的宏只是进行简单的字符替换。

(2) 函数调用是在程序运行时处理的，它需要分配临时的内存单元；而宏展开则是在编译时进行的，在展开时并不分配内存单元，也不进行值的传递处理，也没有“返回值”的概念。

(3) 对函数中的实参和形参都要定义类型，两者的类型要求一致，如果不一致，应进行类型转换；而宏不存在类型问题，宏名无类型，它的参数也无类型，只是一个符号代表，展开时带入指定的字符即可。宏定义时，字符串可以是任何类型的数据。

(4) 调用函数只可得到一个返回值，而用宏可以设法得到几个结果。

(5) 使用宏次数多时，宏展开后源程序会变得很长，因为每展开一次都使程序内容增长，而函数调用不使源程序变长。

(6) 宏替换不占用运行时间，而函数调用则占运行时间（分配单元、保留现场、值传递、返回）。

(7) 参数每次用于宏定义时，它们都将重新求值，由于多次求值，具有副作用的参数可能会产生不可预料的结果。而参数在函数被调用前只求值一次，在函数中多次使用参数并不会导致多种求值过程，参数的副作用并不会造成任何特殊的问题。

一般来说，用宏来代表简短的表达式比较合适。

7.5.6 宏定义平方运算#define SQR(X) X*X 是否正确

执行平方运算的宏定义不正确，会造成错误。首先，以如下程序代码为例进行分析。

```
#include <stdio.h>
#define SQR(X) X*X

int main( )
{
    int a = 21;
    int k = 2;
    int m = 1;
    int b = SQR(k+m);
    int c = SQR(k+m)/SQR(k+m);
```

```

a /= SQR(k+m)/SQR(k+m);
printf("%d\n%d\n%d\n",a,b,c);
return 0;
}

```

程序输出结果:

3
5
7

执行 $SQR(k+m)$ 时,题目的意思是希望执行 $(k+m)*(k+m)$ 操作,但因为宏定义中未能规范表示,导致在执行 $b = SQR(k+m)$ 时,错误地执行为 $k+m*k+m=5$; 在执行 $c = SQR(k+m)/SQR(k+m)$ 时,错误地执行为 $k+m*k+m/k+m*k+m=7$ 。

注意求 a 的时候,宏定义是在预处理的时候进行的, $a /= SQR(k+m)/SQR(k+m)$, 不能先执行 $a = a/SQR(k+m)/SQR(k+m)$, 而应该先计算=右边的值 (/=操作符结合方向: 从右到左), 然后再执行复制操作, 此例中 $a=a/7=3$ 。

程序示例如下:

```

#include <stdio.h>
#define N 3
#define Y(n) ((N+1)*n)
int main( )
{
    int p = Y(5+1);
    int z=2 * (N+Y(5+1));
    printf("%d\n",z);
    return 0;
}

```

程序输出结果:

48

上例中, p 的值为 21, z 的值为 48。 $Y(5+1)=((N+1)*5+1)=21$ 。需要清楚的是, 预处理在编译之前执行文本的替换工作。

程序示例如下:

```

#include <stdio.h>
#define F(a,b) a*b
int main( )
{
    printf("%d\n",F(3+6,8-5));
    return 0;
}

```

程序输出结果:

46

即 $F(3+6,8-5) = 3+6*8-5=46$ 。

例如, $\text{int } i=10, j=10, k=3; k*=i+j$, 应该首先计算 $i+j$ 的值为 20, 然后再计算 k 的值, 所以 k 的值为 60。

7.5.7 不能使用大于、小于、if 语句, 如何定义一个宏来比较两个数 a、b 的大小

如果只是进行简单的比较, 则返回比较结果即可, 宏定义可以写为如下方式:

```

#define check(a,b) (((a)-(b))>=fabs((a)-(b)))? "greater": "smaller"

```

但如果需要返回较大的值，则宏定义可以写为

```
#define MAX(a,b) (abs((a)-(b))==((a)-(b))?(a):(b))
```

虽然#define MAX(a,b) (abs((a)-(b))==((a)-(b))?(a):(b))是一种比较好的做法，但是函数 abs() 接收的参数及其返回值都是整数，这样在传递实参时，其小数部分可能被截去，从而导致误差。例如，a=-12.345，b=-24.1467，abs((a)-(b))返回值为 12，但(a)-(b)显然不等于 12，从而 MAX(a,b)等 b 的值。

#define MAX(a,b)(((a)-(b))&0x80000000)?(b):(a)及#define MAX(a,b)(((b)-(a)&(0x1<<31))>>31)也都只能对整数进行操作。所以将#define MAX(a,b) (abs((a)-(b))==((a)-(b))?(a):(b))中的 abs() 函数换成 fabs()函数，fabs()所接受的参数及返回值都是 double 型的，这样无论它是接受整数还是接受 float 型的数据，都不会因精度问题而出现误差。

引申：写一个“标准”宏 MIN，这个宏输入两个参数并返回较小的一个。

由于此时没有限定，所以可以使用如下方式的宏定义：

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

7.5.8 如何判断一个变量是有符号数还是无符号数

判断一个变量是无符号数还是有符号数有以下 3 种方法：

(1) 采用取反操作。

对于这个变量分两种情况进行分析，一种情况是它为某种类型的值，另一种情况是它为某种类型。对于值而言，如果这个数以及其求反后的值都大于 0，则该数为无符号数，反之则为有符号数，因为数据在计算机中都是以二进制的 0 或 1 存储的，正数以 0 开头，负数以 1 开头，求反操作符会把所有的 1 改为 0，所有的 0 改为 1。如果有符号数，那么取反之后，开头的 0 会被改为 1，开头的 1 会被改为 0，开头为 1 时即表示该数为负数，如果是无符号数则不会受此影响。对于类型而言，也同样适用。

对于为值的情况，可以采用如下宏定义的方式：

```
#define ISUNSIGNED(a) (a>=0 && ~a>=0)
```

对于为类型的情况：

```
#define ISUNSIGNED(type) ((type)0-1 > 0)
```

前者一般只适用于 K&R C，不适用于 ANSI C 的情况。

程序示例代码如下：

```
#include <stdio.h>

#define ISUNSIGNED(a) (a >= 0) && (~a >= 0)
#define ISUNSIGNED_TYPE(type) ((type)0-1 > 0)

int main()
{
    int a = 0;
    unsigned int b = 0;
    printf("%d\n", ISUNSIGNED(a));
    printf("%d\n", ISUNSIGNED(b));
    printf("%d\n", ISUNSIGNED_TYPE(int));
    printf("%d\n", ISUNSIGNED_TYPE(unsigned int));
    return 0;
}
```

程序输出结果：

```
0
1
```

0
1

(2) 由于无符号数和有符号数相减的结果为无符号，所以还可以采用以下方法判断：

```
#include<stdio.h>
int main( )
{
    int a = 100;
    int b = -1;
    if(a<0)
    {
        printf("有符号数");
    }
    else
    {
        if(b-a>0)
            printf("无符号数\n");
        else
            printf("有符号数\n");
    }
    return 0;
}
```

程序输出为

有符号数

上例中，当把变量 a 的类型变为 unsigned int 时，程序的输出则变为

无符号数

(3) 通过改变符号位判断。把 A 进行一个位运算，将最高位置 1，判断是否大于 0。

程序示例如下：

```
#include <stdio.h>

int main( )
{
    unsigned A = 10;
    A = A|(1 << 31);
    if(A > 0)
        printf("无符号数\n");
    else
        printf("有符号数\n");
    return 0;
}
```

程序输出为

有符号数

7.5.9 #define TRACE(S) (printf("%s\n", #S), S)是什么意思

#进行宏字符串连接，在宏中把参数解释为字符串，不可以在语句中直接使用。在宏定义中 printf("%s\n", #S)会被解释为 printf("%s\n", "S")。

程序示例如下：

```
#include <stdio.h>
#include <string.h>
#define TRACE(S) (printf("%s\n", #S), S)
int main( )
{
    int a=5;
    int b=TRACE(a);
}
```



```

    const char *str="hello";
    char des[50];
    strcpy(des,TRACE(str));
    printf("%s\n",des);
    return 0;
}

```

程序输出结果:

```

a
str
hello

```

上例中, 宏定义又是一个逗号表达式, 所以复制到 `des` 里面的值为后面 `S` 也就是 `str` 的值。所以最后输出的就是字符串 `hello` 了。

7.5.10 不使用 `sizeof`, 如何求 `int` 占用的字节数

一般求解字节数, 最常采用的方法是采用 `sizeof` 求解。例如, 在 32 位机器下, `int` 型变量占用的内存空间大小为 4 个字节, 而本题要求不使用 `sizeof`, 所以只能从原理上对 `int` 型变量所占的空间进行求解。

一般可以使用如下的方式实现:

```

#include <stdio.h>

#define MySizeof(Value) (char* )(&Value + 1) - (char* )&Value

int main( )
{
    int i;
    double f;
    double a[4];
    double* q;
    printf("%d\n",MySizeof(i));
    printf("%d\n",MySizeof(f));
    printf("%d\n",MySizeof(a));
    printf("%d\n",MySizeof(q));
    return 0;
}

```

程序的输出结果:

```

4
8
32
4

```

上例中, `(char*)& Value` 返回 `Value` 的地址的第一个字节, `(char*)(& Value +1)` 返回 `Value` 的地址的下一个地址的第一个字节, 所以它们之差为它所占的字节数。

如果不使用宏定义的方式, 也可以使用如下方式求解, 程序示例代码如下:

```

#include <iostream>
using namespace std;

template <class Any>
int LengthofArray(Any* p)
{
    return int(p+1) - int(p);
}

int main( )
{

```

```

int* i;
double* q;
char a[10];
printf("%d\n",LengthofArray(i));
printf("%d\n",LengthofArray(q));
printf("%d\n",LengthofArray(&a));
return 0;
}

```

程序的输出结果:

```

4
8
10

```

7.5.11 如何使用宏求结构体的内存偏移地址

```
#define OffSet(type, field) ((size_t)&((type *)0 -> field))
```

在 C 语言中, ANSI C 标准允许值为 0 的常量被强制转换成任何一种类型的指针, 而且转换结果是一个空指针, 即 NULL 指针, 因此对 0 取指针的操作((type *)0)的结果就是一个类型为 type* 的 NULL 指针。但如果利用这个 NULL 指针来访问 type 的成员当然是非法的, 因为 &(((type *)0)->field) 的意图只不过是计算 field 字段的地址。

C 语言编译器根本就不生成访问 type 的代码, 而仅仅是根据 type 的内容布局 and 结构体实例首址在编译期计算这个 (常量) 地址, 这样就完全避免了通过 NULL 指针访问内存可能出现的问题。同时又因为地址为 0, 所以这个地址的值就是字段相对于结构体基址的偏移。

程序示例如下:

```

#include <stdio.h>
#define OffSet(type,field) ((size_t)&(((type *)0)->field))

struct MyStr
{
    char a;
    int b;
    float c;
    double d;
    char e;
};

int main( )
{
    printf("%d\n",OffSet(MyStr,a));
    printf("%d\n",OffSet(MyStr,b));
    printf("%d\n",OffSet(MyStr,c));
    printf("%d\n",OffSet(MyStr,d));
    printf("%d\n",OffSet(MyStr,e));
    return 0;
}

```

在 32 位机器上, char 型占 1 个字节, int 型占 4 个字节, float 型占 4 个字节, double 占 8 个字节, 所以经过 VC++ 编译运行后, 程序的输出为如下:

```

0
4
8
16
24

```

上述方法避免了实例化一个 type 对象, 而且求值在编译期进行, 没有运行期负担, 程序

效率大大提高。

7.5.12 如何用 sizeof 判断数组中有多少个元素

只需要用整个数组的 sizeof 去除以一个元素的 sizeof 即可求出数组中元素的个数，以数组名 array 为例，代码为 `#define Count (sizeof(array)/sizeof(array[0]))` 或者 `#define Count (sizeof(array)/sizeof(数组的类型, 如 int、double 等))`。程序示例如下：

```
#include <stdio.h>

#define Count (sizeof(array)/sizeof(array[0]))
int main( )
{
    int array[] = {1,2,3,4,5};
    printf("%d\n",Count);
    return 0;
}
```

程序输出结果：

5

之所以以上两种写法都可以，是因为在数组中 `sizeof(array[0])` 的值本质上就是 `sizeof(数组的类型, 如 int、double 等)`，所以两者等价，以上两种形式的计算都可以。

7.5.13 枚举和 define 有什么不同

两者只有很小的区别。在 C 语言中，枚举为整型，枚举常量为 int 型，因此它们都可以和其他整型类别混用而不会出错，而且枚举优点众多：能自动赋值；调试器在检验枚举变量时，可以显示符号值；服从数据块作用域规则。具体而言，两者的区别表现在以下几个方面：

(1) 枚举常量是实体中的一种，而宏定义不是实体。

(2) 枚举常量属于常量，但宏定义不是常量。

(3) 枚举常量具有类型，但宏没有类型，枚举变量具有与普通变量相同的性质，如作用域、值等，但是宏没有。

(4) `#define` 宏常量是在预编译阶段进行简单替换，枚举常量则是在编译的时候确定其值。

(5) 一般在编译器里，可以调试枚举常量，但是不能调试宏常量。

(6) 枚举可以一次定义大量相关的常量，而 `#define` 宏一次只能定义一个。

7.5.14 typedef 和 define 有什么区别

`typedef` 与 `define` 都是替一个对象取一个别名，以此来增强程序的可读性，但是它们在使用和作用上也存在着以下几个方面的不同：

(1) 原理不同。`#define` 是 C 语言中定义的语法，它是预处理指令，在预处理时进行简单而机械的字符串替换，不作正确性检查，不管含义是否正确照样带入，只有在编译已被展开的源程序时才会发现可能的错误并报错。

例如，`#define PI 3.1415926`，当程序中执行 `area=PI*r*r` 语句时，PI 会被替换为 3.1415926，于是该语句被替换为 `area=3.1415926*r*r`。如果把 `#define` 语句中的数字 9 写成了 g，预处理也照样带入，而不去检查其是否合理、合法。

`typedef` 是关键字，它在编译时处理，所以 `typedef` 有类型检查的功能。它在自己的作用

域内给一个已经存在的类型一个别名，但是不能在一个函数定义里面使用标识符 `typedef`。例如，`typedef int INTEGER`，这以后就可用 `INTEGER` 来代替 `int` 作整型变量的类型说明了，如：

```
INTEGER a, b;
```

用 `typedef` 定义数组、指针、结构等类型将带来很大的方便，不仅使程序书写简单而且使意义更为明确，因而增强了可读性。例如：

```
typedef int a[10];
```

表示 `a` 是整型数组类型，数组长度为 10。然后就可用 `a` 说明变量，如 `a s1,s2`；完全等效于 `int s1[10],s2[10]`。同理，`typedef void (*p)(void)` 表示 `p` 是一种指向 `void` 型的指针类型。

(2) 功能不同。`typedef` 用来定义类型的别名，这些类型不只包含内部类型 (`int`、`char` 等)，还包括自定义类型 (如 `struct`)，可以起到使类型易于记忆的功能。

例如：`typedef int (*PF)(const char *, const char *)`;

定义一个指向函数的指针的数据类型 `PF`，其中函数返回值为 `int`，参数为 `const char *`。`typedef` 还有另外一个重要的用途，那就是定义机器无关的类型。例如，可以定义一个叫 `REAL` 的浮点类型，在目标机器上它可以获得最高的精度：`typedef long double REAL`，在不支持 `long double` 的机器上，该 `typedef` 看起来会是下面这样：`typedef double REAL`，在连 `double` 都不支持的机器上，该 `typedef` 看起来会是这样：`typedef float REAL`。

`#define` 不只是可以为类型取别名，还可以定义常量、变量、编译开关等。

(3) 作用域不同。`#define` 没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用，而 `typedef` 有自己的作用域。

程序示例如下：

```
void fun()
{
    #define A int
}
void gun()
{
    //在这里也可以使用 A，因为宏替换没有作用域，但如果上面用的是 typedef，那这里就不能用
    //A，不过一般不在函数内使用 typedef
}
```

(4) 对指针的操作不同。两者修饰指针类型时，作用不同。

```
#define INTPTR1 int*
typedef int* INTPTR2;
INTPTR1 p1,p2;
INTPTR2 p3,p4;
```

`INTPTR1 p1,p2` 和 `INTPTR2 p3,p4` 这两句的效果截然不同的。`INTPTR1 p1,p2` 进行字符串替换后变成 `int* p1,p2`，要表达的意义是声明一个指针变量 `p1` 和一个整型变量 `p2`。而 `INTPTR2 p3,p4`，由于 `INTPTR2` 是具有含义的，告诉我们是一个指向整型数据的指针，那么 `p3` 和 `p4` 都为指针变量，这句相当于 `int* p1,*p2`。从这里可以看出，进行宏替换是不含任何意义的替换，仅仅为字符串替换；而用 `typedef` 为一种数据类型起的别名是带有一定含义的。

程序示例如下：

```
#define INTPTR1 int*
typedef int* INTPTR2;
int a=1;
```



```
int b=2;
int c=3;
const INTPTR1 p1=&a;
const INTPTR2 p2=&b;
INTPTR2 const p3=&c;
```

上述代码中, `const INTPTR1 p1` 表示 `p1` 是一个常量指针, 即不可以通过 `p1` 去修改 `p1` 指向的内容, 但是 `p1` 可以指向其他内容。而对于 `const INTPTR2 p2`, 由于 `INTPTR2` 表示是一个指针类型, 因此用 `const` 去限定, 表示封锁了这个指针类型, 因此 `p2` 是一个指针常量, 不可使 `p2` 再指向其他的内容, 但可以通过 `p2` 修改其当前指向的内容。`INTPTR2 const p3` 同样声明的是一个指针常量。

7.5.15 C++中宏定义与内联函数有什么区别

宏代码本身不是函数, 但使用起来却像函数, 预处理器用复制宏代码的方式代替函数调用, 省去了参数压栈、生成汇编语言的 `CALL` 调用、返回参数、执行 `return` 等过程, 从而提高了速度。内联函数是代码被插入到调用者代码处的函数。对于 C++ 而言, 内联函数 (`inline`) 的作用也不是万能的, 它的使用是有所限制的, 它只适合函数体内代码简单的函数使用, 不能包含复杂的结构控制语句 (如 `while`、`switch`), 并且内联函数本身不能直接调用递归函数 (自己内部还调用自己的函数)。

两者的区别主要表现在以下几个方面: 第一, 宏定义是在预处理阶段进行代码替换, 而内联函数是在编译阶段插入代码; 第二, 宏定义没有类型检查, 而内联函数有类型检查。

引申: 内联函数与普通函数的区别有哪些?

内联函数的参数传递机制与普通函数相同, 但是编译器会在每处调用内联函数的地方将内联函数的内容展开, 这样既避免了函数调用的开销又没有宏机制的缺陷。

内联函数和普通函数最大的区别在于其内部的实现方面上, 普通函数在被调用时, 系统首先要跳跃到该函数的入口地址, 执行函数体, 执行完成后, 再返回到函数调用的地方, 函数始终只有一个复制; 而内联函数则不需要进行一个寻址的过程, 当执行到内联函数时, 此函数展开, 如果在 `N` 处调用了此内联函数, 则此函数就会有 `N` 个代码段的复制。

内联函数也并非万金油, 在使用的过程中也存在一定的局限性, 如果函数体过大, 编译器也会放弃内联方式, 而采用普通的方式进行调用函数。此时, 内联函数就和普通函数执行效率一样了。

7.5.16 定义常量谁更好? #define 还是 const

尺有所短, 寸有所长, `define` 与 `const` 都能定义常量, 效果虽然一样, 但是各有侧重。`define` 既可以替代常数值, 又可以替代表达式, 甚至是代码段, 但是容易出错, 而 `const` 的引入可以增强程序的可读性, 它使程序的维护与调试变得更加方便。具体而言, 它们的差异主要表现在以下几个方面:

(1) `define` 只是用来进行单纯的文本替换, `define` 常量的生命周期止于编译期, 不分配内存空间, 它存在于程序的代码段, 在实际程序中它只是一个常数, 一个命令中的参数并没有实际的存在; 而 `const` 常量存在于程序的数据段, 并在堆栈中分配了空间, `const` 常量在程序中确实确实地存在, 并且可以被调用、传递。

(2) `const` 常量有数据类型, 而 `define` 常量没有数据类型。

编译器可以对 `const` 常量进行类型安全检查，如类型、语句结构等，而 `define` 不行。

(3) 很多 IDE 支持调试 `const` 定义的常量，而不支持 `define` 定义的常量。

由于 `const` 修饰的变量可以排除程序之间的不安全性因素，保护程序中的常量不被修改，而且对数据类型也会进行相应的检查，极大地提高了程序的健壮性，所以一般更加倾向于用 `const` 来定义常量类型。

7.6 结构体与类

不要因为 C 和 C++ 中有一些语法和关键字看上去相同，就认为它们的意义和作用完全一样。有些时候它们是不一样的，如 `struct` (结构体) 与 `class` (类)。

7.6.1 C 语言中 `struct` 与 `union` 的区别是什么

`struct` (结构体) 与 `union` (联合体) 是 C 语言中两种不同的数据结构，两者都是常见的复合结构，其区别主要表现在以下两个方面：

(1) 结构体与联合体虽然都是由多个不同的数据类型成员组成的，但不同之处在于联合体中所有成员共用一块地址空间，即联合体只存放了一个被选中的成员，而结构体中所有成员占用空间是累加的，其所有成员都存在，不同成员会存放在不同的地址。在计算一个结构型变量的总长度时，其内存空间大小等于所有成员长度之和 (需要考虑字节对齐)，而在联合体中，所有成员不能同时占用内存空间，它们不能同时存在，所以一个联合型变量的长度等于其最长的成员的长度。

(2) 对于联合体的不同成员赋值，将会对它的其他成员重写，原来成员的值就不存在了，而对结构体的不同成员赋值是互不影响的。

例如：`typedef union {double i; int k[5]; char c;} DATE;`

`struct data { int cat; DATE cow; double dog;} too;`

`DATE max;` 则语句 `printf("%d",sizeof(struct data)+sizeof(max));` 的执行结果是多少？

假设为 32 位机器，`int` 型占 4 个字节，`double` 占 8 个字节，`char` 型占 1 个字节，而 `DATE` 是一个联合型变量，联合型变量公用空间，里面最大的变量类型是 `int[5]`，所以占用 20 个字节，它的大小是 20，而 `data` 是一个结构体变量，每个变量分开占用空间，依次为 `sizeof(int)+sizeof(DATE)+sizeof(double)=4+20+8=32`，所以结果是 $32 + 20 = 52$ 个字节。

7.6.2 C 和 C++ 中 `struct` 的区别是什么

C 语言中的 `struct` 与 C++ 中的 `struct` 的区别表现在以下 3 个方面：

(1) C 语言的 `struct` 不能有函数成员，而 C++ 的 `struct` 可以有。

(2) C 语言的 `struct` 中数据成员没有 `private`、`public` 和 `protected` 访问权限的设定，而 C++ 的 `struct` 的成员有访问权限设定。

(3) C 语言的 `struct` 是没有继承关系的，而 C++ 的 `struct` 却有丰富的继承关系。

C 语言中的 `struct` 是用户自定义数据类型 (User Defined Type)，它是没有权限设置的，它只能是一些变量的集合体，虽然可以封装数据却不可以隐藏数据，而且成员不可以是函数。为了和 C 语言兼容，C++ 中就引入了 `struct` 关键字。C++ 语言中的 `struct` 是抽象数据类型 (ADT)，它支持成员函数的定义，同时它增加了访问权限，它的成员函数默认访问权限为 `public`。在用模板的时候只能写 `template <class Type>` 或 `template <typename Type>` 不能写 `template <struct Type>`。

7.6.3 C++中 struct 与 class 的区别是什么

如果没有多态和虚拟继承,在 C++中,struct 和 class 的存取效率完全相同,存取 class 的数据成员与非虚函数效率和 struct 完全相同,不管该数据成员是定义在基类还是派生类的。

class 的数据成员在内存中的布局不一定是数据成员的声明顺序,C++只保证处于同一个 access section 的数据成员按照声明顺序排列。

具体而言,在 C++中,class 和 struct 做类型定义时只有两点区别:首先是默认继承权限,class 继承默认是 private 继承,而 struct 继承默认是 public 继承;其次是 class 还用于定义模板参数,就像 typename,但关键字 struct 不用于定义模板参数。

C++中之所以保留 struct 关键字,主要有 3 个方面的原因:第一,保证与 C 语言的向下兼容性,C++必须提供一个 struct;第二,C++中的 struct 定义必须百分之百地保证与 C 语言中的 struct 的向下兼容性,之所以把 C++中最基本的对象单元规定为 class 而不是 struct,就是为了能够避免各种兼容性要求的限制;第三,对 struct 定义的扩展使 C 语言代码能够更容易地被移植到 C++中来。

7.7 位操作

二进制是现代计算机发展的基础,所有的程序代码都需要转换成最终的二进制代码才能执行。合理地进行二进制的位操作,对于编写优质代码,特别是嵌入式应用软件开发非常关键。

7.7.1 一些结构声明中的冒号和数字是什么意思

C 语言的结构体可以实现位段,它的定义形式是在一个定义的结构体成员后面加上冒号,然后是该成员所占的位数。位段的结构体成员必须是 int 或者 unsigned int 类型,不能是其他类型。位段在内存中的存储方式是由具体的编译器决定的。

首先,定义位段的长度不能大于存储单元的长度。存储单元是指该位段的类型大小,不是计算机的存储单元字节。其次,一个位段如果不能放在一个存储单元里,那么它会把这个存储单元中剩余的空间闲置,而从下一个存储单元开始存储下一个位段,即一个位段不能存储在两个存储单元内,位段在一个存储单元中的存储是紧凑的。再次,位段名缺省时称作无名位段,无名位段的存储空间通常不用,而位段长度为 0 位表示下一个位段存储在一个新的存储单元中,位段长度为 0 的时候位段名必须缺省(不能定义位段名)。最后,一个结构体中既可以定义位段成员也可以同时定义一般的结构体成员。这个时候,一般成员不和位段存储在同一个存储单元中。

程序示例分析如下:

```
#include <stdio.h>
```

```
typedef struct
{
    int a:2;
    int b:2;
    int c:1;
}test;
```

```
int main()
{
    test t;
    t.a = 1;
```

```

t.b = 3;
t.c = 1;
printf("%d\n%%d\n%d\n",t.a, t.b, t.c);
return 0;
}

```

程序输出结果:

```

1
-1
-1

```

由于 a 占两位, 而 a 被赋值为 1, 二进制就是 01, 因此 %d 输出的时候输出 1; b 也占了两位, 赋值为 3, 二进制也就是 11, 由于使用了 %d 输出, 表示的是将这个 b 作为有符号 int 型来输出, 这样的话二进制的 11 将会有一位被认为是符号位, 并且两位的 b 也会被扩展为 int 类型, 也就是 4 字节, 即 32 位。其实 a 也做了这种扩展, 只是扩展符号位的时候, 由于数字在计算机中存储都是补码形式, 因此扩展符号位的时候正数用 0 填充高位, 负数则用 1 填充高位。因此对于 a 来说, 输出的时候被扩展为 00000000 00000000 00000000 00000001, 也就是 1, 而 b 则扩展为 11111111 11111111 11111111 11111111, 也就是 -1 了, c 的显示也是这样的。

7.7.2 最有效的计算 2 乘以 8 的方法是什么

$2 \ll 3$ 。

虽然直接进行乘法操作符运算也可以进行 2 与 8 的相乘, 但是该方法并非最优, 通过移位方法会比较高效。因为将一个数左移 n 位, 相当于乘以了 2 的 n 次方。因此, 一个数乘以 8, 而 8 是 2 的 3 次方, 所以只要该数左移 3 位即可实现乘以 8 的目的。

常规的乘法运算也可以实现, 但 CPU 直接支持位运算, 效率最高, 所以操作 2 乘以 8 的最有效的方法是 $2 \ll 3$ 。

引申: 如何快速求取一个整数的 7 倍?

相比移位运算, 如果直接使用乘法运算符的话, 则执行效率相对比较慢, 所以快速的方法就是将这个乘法转换成加减法和移位操作。由于移位运算相当于乘法运算或除法运算, 左移相当于乘法运算, 右移运算相当于除法运算, 所以此时可以先将此整数左移 3 位 (相当于将数字乘以 8), 然后再减去原值, 即 $(X \ll 3) - X$ 就获得了 X 的 7 倍。此处需要注意的是, 由于 - 的优先级高于 \ll , 所以不能去掉括号, 否则结果不正确。

7.7.3 如何实现位操作求两个数的平均值

一般而言, 求解平均数的方法就是将两者相加, 然后除以 2, 以变量 x 与 y 为例, 两者的平均数为 $(x+y)/2$ 。

但是采用上述方法, 会存在一个问题, 当两个数比较大时, 如两者的和大于了机器位数能够表示的最大值, 可能会存在数据溢出的情况, 而采用位运算方法则可以避免这一问题, $(x \& y) + ((x \wedge y) \gg 1)$ 方式表达的意思都是求解变量 x 与 y 的平均数, 而且位运算相比除法运算, 效率更高。

对于表达式 $(x \& y) + ((x \wedge y) \gg 1)$, $x \& y$ 表示的是取出 x 与 y 二进制位数中都为 '1' 的所有位, $x \wedge y$ 表示的是 x 与 y 中有一个为 '1' 的所有位, 右移 1 位相当于执行除以 2 运算。整个表达式实际上可以分为两部分, 第一部分是都为 '1' 的部分, 因为相同, 所以直接相加即可; 而第二部分是 x 为 '1'、y 为 '0' 的部分, 以及 y 为 '1'、x 为 '0' 的部分, 两部分加起来再除以 2, 然后跟前面的相加就可以表示两者的平均数了。

以下述示例为例。

```
#include<stdio.h>

int main( )
{
    int x = 2147483647, y = 2147483647;
    printf("%d\n", (x+y)/2);
    printf("%d\n", (x&y)+((x^y)>>1));
    return 0;
}
```

在 32 位机器下，程序输出结果如下：

```
-1
2147483647
```

程序的输出正好验证了这一算法的可行性。

引申：如何利用位运算计算数的绝对值？

以 x 为负数为例来分析。因为在计算机中，数字都是以补码的形式存放的，求负数的绝对值，应该是不管符号位，执行按位取反，末位加 1 操作即可。

对于一个负数，将其右移 31 位后会变成 $0xffffffff$ ，而对于一个正数而言，右移 31 位则为 $0x00000000$ ，而 $0xffffffff \wedge x + x = -1$ ，因为 $1011 \wedge 1111 = 0100$ ，任何数与 1111 异或，其实质都是把 x 的 0 和 1 进行颠倒计算。如果用变量 y 表示 x 右移 31 位， $(x \wedge y) - y$ 则表示的是 x 的绝对值。

程序示例如下：

```
#include<stdio.h>

int MyAbs( int x)
{
    int y;
    y = x >> 31;
    return (x^y)-y; //此处还可以写为(x+y)^y
}

int main( )
{
    printf("%d\n", MyAbs(2));
    printf("%d\n", MyAbs(-2));
    return 0;
}
```

程序输出结果：

```
2
2
```

上例中，在函数 `MyAbs` 中，对局部变量 y 进行赋值时，由于是对 x 进行右移 31 位，如果 x 为正数，则 $y = 0$ ；如果 x 为负数，则 $y = -1$ 。

7.7.4 unsigned int i=3; printf("%u\n",i*-1)输出为多少

运行如下程序：

```
#include <stdio.h>

int main( )
{
    unsigned int i=3;
```

```
printf("%u\n", i*-1);
return 0;
```

程序输出结果:

4294967293

在 32 位机器中, $i*-1$ 的值为 4294967293。在 32 位机器中, 无符号 int 的值域是 $[0, 4294967295]$, 有符号 int 的话, 值域是 $[-2147483648, 2147483647]$, 两个值域的个数都是 4294967296 个, 即

$[0, 4294967295] = [0, 2147483647] \cup [2147483648, 4294967295]$

有符号 int 的 $[-2147483648, -1]$ 对应于无符号 int 的 $[2147483648, 4294967295]$ 区域, 两个区域的值是一一映射关系。所以, -1 对应 4294967295, -2 对应 4294967294, -3 对应 4294967293。

引申: unsigned short A = 10; printf("~A = %u\n", ~A); 输出是什么?

因为 A 为无符号短整型变量, 值为 10, 在 32 位机器中, 转换为二进制为 0000 0000 0000 0000 0000 0000 0000 1010, 对 A 取反操作, 所以 ~A 的二进制位为 1111 1111 1111 1111 1111 1111 1111 0101, 十六进制表示即为 0xFFFFFFFF5, 而如果将该数转换为符号整型的话则为 -11, 因为输出的是无符号整型, 无符号整型的范围为 0~4294967295, 而 0xFFFFFFFF5 转换为无符号十进制整型为 4294967285。

所以程序的输出结果为 4294967285。

7.7.5 如何求解整型数的二进制表示中 1 的个数

求解整型数的二进制表示中 1 的个数有以下两种方法:

方法一, 程序代码如下:

```
#include<stdio.h>

int func(int x)
{
    int countx = 0;
    while(x)
    {
        countx++;
        x = x&(x-1);
    }
    return countx;
}

int main()
{
    printf("%d\n", func(9999));
    return 0;
}
```

程序输出结果:

8

在上例中, 函数 func() 的功能是将 x 转化为二进制数, 然后计算该二进制数中含有的 1 的个数。首先以 9 为例来分析, 9 的二进制为 1001, 8 的二进制为 1000, 两者执行 & 操作之后结果为 1000, 此时 1000 再与 0111 (7 的二进制位) 执行 & 操作之后结果为 0。

为了理解这个算法的核心, 需要理解以下两个操作:

(1) 当一个数被减 1 时，它最右边的那个值为 1 的 bit 将变为 0，同时其右边的所有的 bit 都会变成 1。

(2) “&=”，位与并赋值操作。去掉已经被计数过的 1，并将该值重新设置给 n。这个算法循环的次数是 bit 位为一的个数。也就是说，有几个 bit 为 1，循环几次，对 bit 为 1 比较稀疏的数来说，性能很好。例如，0x1000 0000 循环一次就可以。

方法二，判断每个数的二进制表示中每一位是否为 1，如果为 1，就在 count 上加 1，而循环的次数是常数，即 n 的位数。但该方法有一个缺陷，就是在 1 比较稀疏的时候效率会比较低。

程序示例如下：

```
#include<stdio.h>

int func (int n)
{
    int count=0;
    while (n)
    {
        count += n & 0x1u;
        n >>= 1;
    }
    return count;
}

int main( )
{
    printf("%d\n", func(9999));
    return 0;
}
```

程序输出结果：

8

需要注意的是，上例中，0x1u 表示的是 16 进制的无符号数 1。

7.7.6 不能用 sizeof() 函数，如何判断操作系统是 16 位还是 32 位的

如果没有强调不许使用 sizeof，一般可以使用 sizeof 计算字节长度来判断操作系统的位数，如在 32 位机器上，sizeof(int) = 4，而在 16 位机器上，sizeof(int)=2。除此之外，还有以下两种方法。

方法一：一般而言，机器位数不同，其表示的数字的最大值也不同，根据这一特性，可以判断操作系统的位数。

例如，运行如下代码。

```
#include <stdio.h>

int main( )
{
    int i = 65536;
    printf("%d\n",i);
    int j = 65535;
    printf("%d\n",j);
    return 0;
}
```

由于 16 位机器下，无法表示这么大的数，会出现越界情况，所以程序输出为

0

-1

而在 32 位机器下，则会正常输出，程序输出为

```
65536
65535
```

之所以会有区别，是因为在 16 位机器下，能够表示的最大数为 65535，所以会存在最高位溢出的情况。当变量的值为 65536 时，输出为 0；当变量的值为 65535 时，输出为-1。而在 32 位机器上，则不会出现溢出的情况，所以输出为正常输出。

方法二：对 0 值取反，不同位数下的 0 值取反，其结果不一样。例如，在 32 位机器下，按位取反运算，结果为 11111111111111111111111111111111。运行如下代码：

```
#include <stdio.h>
```

```
int main( )
{
    unsigned int a = ~0;
    if( a>65536 )
        printf("32 位\n");
    else
        printf("16 位\n");
    return 0;
}
```

程序输出为
32 位

7.7.7 嵌入式编程中，什么是大端？什么是小端

采用小端模式的 CPU 对操作数的存放方式是从低字节到高字节，而大端模式对操作数的存放方式是从高字节到低字节。例如，16 位宽的数 0x1234 在小端模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）见表 7-4，而在大端模式 CPU 内存中的存放方式见表 7-5。

表 7-4 0x1234 在小端模式 CPU 内存中的存放方式

| 内存地址 | 存放内容 |
|--------|------|
| 0x4000 | 0x34 |
| 0x4001 | 0x12 |

表 7-5 0x1234 在大端模式 CPU 内存中的存放方式

| 内存地址 | 内存内容 |
|--------|------|
| 0x4000 | 0x12 |
| 0x4001 | 0x34 |

32 位宽的数 0x12345678 在小端模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）见表 7-6，而在大端模式 CPU 内存中的存放方式见表 7-7。

表 7-6 0x12345678 在小端模式 CPU
内存中的存放方式

| 内存地址 | 存放内容 |
|--------|------|
| 0x4000 | 0x78 |
| 0x4001 | 0x56 |
| 0x4002 | 0x34 |
| 0x4003 | 0x12 |

表 7-7 0x12345678 在大端模式 CPU
内存中的存放方式

| 内存地址 | 存放内容 |
|--------|------|
| 0x4000 | 0x12 |
| 0x4001 | 0x34 |
| 0x4002 | 0x56 |
| 0x4003 | 0x78 |

以如下程序为例。

```
#include <stdio.h>

struct mybitfields
```



```
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
}test;

int main( )
{
    int i;
    test.a=2;
    test.b=3;
    test.c=0;
    i=((short *)&test);
    printf("%d\n",i);
    return 0;
}
```

程序输出结果:

50

上例中 sizeof(test)=16，上例的声明方式是把一个 short（也就是一块 16 位内存）分成 3 部分，各部分的大小分别是 4 位、5 位、7 位，赋值语句 i=((short *)&test)就是把上面的 16 位内存转换成 short 类型进行解释。

变量 a 的二进制表示为 0000000000000010，取其低四位是 0010。变量 b 的二进制表示为 0000000000000011，取其低五位是 00011。变量 c 的二进制表示为 0000000000000000，取其低七位是 0000000。

x86 机是小端（修改分区表时要注意）模式，单片机一般为大端模式。小端一般是低位字节在高位字节的前面，也就是低位在内存地址低的一端，可以这样记（小端→低位→在前→与正常逻辑顺序相反），所以合成后得到 0000000000110010，即十进制的 50。

程序示例如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main( )
{
    unsigned int uiVal_1 = 0x12345678;
    unsigned int uiVal_2 = 0;
    unsigned char aucVal[4] = {0x12, 0x34, 0x56, 0x78};
    unsigned short usVal_1 = 0;
    unsigned short usVal_2 = 0;
    memcpy(&uiVal_2, aucVal, sizeof(uiVal_2));
    usVal_1 = (unsigned short)uiVal_1;//在这儿截断，都取得的是低位
    usVal_2 = (unsigned short)uiVal_2;//在这儿截断
    printf("usVal_1: %x\n", usVal_1);//这儿又转化回来
    printf("usVal_2: %x\n", usVal_2);//这边又转化回来
    return 0;
}
```

小端模式是低地址存放低字节，高地址存放高字节，结构如图 7-4 所示。

在内存里面测试机是小端，地址由小到大。

```
uiVal_1: 78 56 34 12
uiVal_2: 12 34 56 78
```

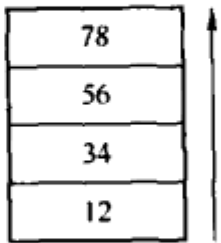


图 7-4 小端模式的存放方式

结果如下所示。

```
5678
3412
```

再例如：

```
char p[4]={0x01,00,0x01,00};
printf("%d",*(int*)p);
```

内存布局如下所示。

```
0x01
0x00
0x01
0x00
```

如果用 `char*` 指针访问，那么一个元素占一个字节；如果用 `int*` 指针访问，那么一个元素占 4 个字节。这样又涉及大小端的问题。

小端的时候把数据解释成 `0x00010001`，所以是 65537。

大端的时候把数据解释成 `0x01000100`。

程序实例如下：

```
#include<stdio.h>
#include<string.h>

typedef struct  AA
{
    int b1:5;
    int b2:5;
}AA;

int main( )
{
    AA aa;
    char cc[100];
    strcpy(cc,"0123456789abcdefghijklmnopqrstuvwxyz");
    memcpy(&aa,cc,sizeof(AA));
    printf("%d\n",aa.b1);
    printf("%d\n",aa.b2);
    printf("%d\n",sizeof(aa));
    return 0;
}
```

上述代码输出结果：

```
-16
9
4
```

首先看 `sizeof` 是 4，就是如果出现 `int`，至少为 4 的倍数，如果只有 `char`，就以 `char` 为准。字符 0 的 ASCII 为 48，所以为 `00110000`；字符 1 的 ASCII 为 49，所以为 `00110001`，然后考虑到大端小端，就是 `0000 1100 1000 1100`，前 5 位为 `b1`，反过来，就是 `10000`，由于是补码，所以为 -16，接着的 5 位为 `01001`，为 9。

引申：如何判断计算机处理器是大端还是小端？

程序示例如下所示：

```
#include <stdio.h>

int checkCPU( )
{
    {
```

```

        union w
        {
            int a;
            char b;
        } c;
        c.a = 1;
        return (c.b == 1);
    }

int main( )
{
    if (checkCPU( ))
        printf("小端\n");
    else
        printf("大端\n");
    return 0;
}

```

编者的处理器为 Intel 处理器，因为 Intel 处理器一般都是小端模式，所以此时输出为：

小端

上述代码中，如果处理器是大端的，则返回 0；如果是小端，则返回 1。联合体 union 的存放顺序是所有成员都从低地址开始存放，如果能够通过改代码知道 CPU 对内存采用小端还是大端模式读写，一定会令面试官刮目相看。

还可以通过指针地址来判断，由于在 32 位计算机系统中，short 占两个字节，char 占 1 个字节，所以可以采用如下做法实现该判断。

```

#include <stdio.h>

int checkCPU( )
{
    unsigned short usData = 0x1122;
    unsigned char *pucData = (unsigned char*)&usData;
    return (*pucData == 0x22);
}

int main( )
{
    if (checkCPU( ))
        printf("小端\n");
    else
        printf("大端\n");
    return 0;
}

```

程序输出为

小端

7.7.8 考虑 n 位二进制数，有多少个数中不存在两个相邻的 1

当 $n=1$ 时，满足条件的二进制数为 0、1，一共两个数；当 $n=2$ 时，满足条件的二进制数有 00、01、10，一共 3 个数；当 $n=3$ 时，满足条件的二进制数有 000、001、010、100、101，一共 5 个数。对 n 位二进制数，设所求结果为 $a(n)$ ，对于第 n 位的值，分为 0 或者 1 两种情况：

(1) 第 n 位为 0，则有 $a(n-1)$ 个数。

(2) 第 n 位为 1, 则满足没有两个相邻为 1 的条件, 第 $n-1$ 位为 0, 有 $a(n-2)$ 个数, 因此得到结论 $a(n) = a(n-1) + a(n-2)$ 。

通过观察 (2) 中的表达式可以发现, 式子满足斐波拉契数列, 而求解斐波拉契数列一般有两种方法: 递归方法与非递归方法, 本题只将递归的方法写出来, 有兴趣的读者可以自己编写非递归的斐波拉契数列求解方法。

程序代码示例如下:

```
#include<stdio.h>

long Fibonacci(int i)
{
    if(i==1||i==2)
        return 1;
    else
        return(Fibonacci(i-1)+Fibonacci(i-2));
}

int main( )
{
    printf("%ld\n",Fibonacci(7));
    return 0;
}
```

程序输出结果:

13

7.7.9 不用除法操作符如何实现两个正整数的除法

在回答本问题前, 先学习一些有关位运算的知识。

(1) 常用的等式: $-n = \sim(n-1) = \sim n + 1$ 。

(2) 获取整数 n 的二进制中最后一个 1: $n \& (-n)$ 或者 $n \& \sim(n-1)$ 。例如, $n=010100$, 则 $-n=101100$, $n \& (-n)=000100$ 。

(3) 去掉整数 n 的二进制中最后一个 1: $n \& (n-1)$, 如 $n=010100$, $n-1=010011$, $n \& (n-1)=010000$ 。

一般求解两个正整数的除法问题时, 首先考虑到的就是采用除法运算, 但是除了除法运算外, 还有其它方法可以执行除法运算。

方法一, 可以根据除法运算的原理进行减法操作, 对除数循环减被除数, 减一次结果加一, 直到刚好减为 0 或余数小于被除数为止。程序示例如下:

```
#include <stdio.h>

int div(int a,int b)
{
    int result=0;
    if (b == 0)
    {
        printf("除数不能为 0\n");
        return result;
    }
    while(a>=b)
    {
        result++;
        a=a-b;
    }
}
```



```

    }
    return result;
}

int main( )
{
    printf("%d\n",div(10,3));
    return 0;
}

```

程序输出结果:

3

这个算法每次都以一倍的被除数进行叠加, 算法效率并不高, 尤其是当 a 很大, b 很小时, 效率会非常低。

方法二, 递归法求解。以 $100/3$ 为例, 方法一提出的方法分别比较 97, 94, 91, ..., 4, 1, -2, 最后余数为-2, 退出 while 循环, 整个算法需要比较 34 次。如果每次采用将比较数翻倍的比较方法, 则算法效率能够得到极大优化。

程序示例如下所示:

```

#include <stdio.h>

int MyDiv(int a, int b)
{
    int k = 0;
    int c = b;
    int res = 0;
    if (b == 0)
        printf("除数不能为 0\n");
    if (a < b)
        return 0;
    for ( ; a >= c; c <= 1, k++)
        if (a - c < b)
            return 1<<k;
    return MyDiv(a - (c>>1), b) + (1<<(k - 1));
}

int main( )
{
    printf("%d\n",MyDiv(100,3));
    return 0;
}

```

程序输出结果:

33

方法三: 采用移位操作实现, 位操作的效率一般都比较高效。程序示例如下:

```

#include <stdio.h>

int div(const int x, const int y)
{
    int left_num = x;
    int result = 0;
    while (left_num >= y)
    {
        int multi = 1;
        while (y * multi <= (left_num >> 1))
        {

```

```

        multi = multi << 1;
    }
    result += multi;
    left_num -= y * multi;
}
return result;
}

```

```

int main( )
{
    printf("%d\n",div(10,3));
    return 0;
}

```

程序输出结果:

3

引申 1: 如何只用逻辑运算实现加法运算?

实现两个正整数相加,一般直接使用加号运算符即可。考虑到题目中的要求,与上例中的方法二类似,也可以通过移位操作符来进行正整数的加法运算。例如,5与7求和,转换为二进制求和为101与111求和,其二进制结果为1100。对于二进制的加法而言,1+1=0,1+0=1,0+1=1,0+0=0,通过对比位运算中的异或方法,不难发现,此方法与位运算中的异或类似。那么第一个数值就是:tempNum1 = num1 ^ num2; 0+0的进位是0,1+0的进位是0,只有1+1的进位有效,该思路与位运算的&运算相似,即只有1&1=1,所以可以先num1& num2,由于进位是进到高一位的,与<<运算很相似,同时 num1 和 num2 相互&之后,如果结果为0,那么就不存在进位,运算完成,所以可以用递归的思想实现。程序示例如下所示:

```

#include <stdio.h>

int add(int num1, int num2)
{
    if( 0 == num2 )
        return num1;
    int sumTemp = num1 ^ num2;
    int carry = (num1 & num2) << 1;
    return add( sumTemp, carry );
}

int main( )
{
    printf("%d\n",add(100,200));
    return 0;
}

```

程序输出结果:

300

将递归思想转换为非递归思想之后,就成了另外一种思路,程序示例如下:

```

#include <stdio.h>

int add(int num1, int num2)
{
    int sum=0;
    int num3=0;
    int num4=0;
    while((num1&num2)>0)

```

```

    {
        num3=num1^num2;
        num4=num1&num2;
        num1=num3;
        num2=num4<<1;
    }
    sum=num1^num2;
    return sum;
}

int main( )
{
    printf("%d\n",add(100,200));
    return 0;
}

```

程序输出结果：

300

引申 2：如何只用逻辑运算实现乘法运算？

先看一个实例：1011*1010，因为二进制运算的特殊性，可以将该乘法运算表达式拆分为两个运算，1011*0010 与 1011*1000 的和，而对于二进制的运算，左移 1 位，等价于乘以 0010，左移 3 位，等价于乘以 1000，所以两者的乘积为 10110 与 1011000 的和，即为 1101110。

因而乘法可以通过一系列移位和加法完成。最后一个 1 可通过 $b \& \sim(b-1)$ 求得，可通过 $b \& (b-1)$ 去掉，为了高效地得到左移的位数，可提前计算一个 map，算法如下所示：

```

#include <iostream>
#include <map>
using namespace std;

int multiply(int a, int b)
{
    bool neg = (b < 0);
    if(b < 0)
        b = -b;
    int sum = 0;
    map<int, int> bit_map;
    for(int i = 0; i < 32; i++)
        bit_map.insert(pair<int, int>(1 << i, i));
    while(b > 0)
    {
        int last_bit = bit_map[b & ~ (b - 1)];
        sum += (a << last_bit);
        b &= b - 1;
    }
    if(neg)
        sum = -sum;
    return sum;
}

int main( )
{
    printf("%d\n",multiply(3,5));
    return 0;
}

```

程序输出结果：

15

7.8 函数

函数是程序的基本组成单位，利用函数，不仅能够实现程序的模块化，而且简单、直观，能极大地提高程序的易读性和可维护性，所以将程序中的一些计算或操作抽象成函数以供随时调用，理解函数的执行原理以及应用是一个优秀程序员应该具备的基本能力。

7.8.1 怎么样写一个接受可变参数的函数

C 语言中支持函数调用的参数为变参形式。例如，`printf()` 这个函数，它的函数原型是 `int printf(const char* format, ...)`，它除了有一个参数 `format` 固定以外，后面跟的参数的个数和类型都是可变的，可以有以下几种不同的调用方法：

- (1) `printf("%d",i);`
- (2) `printf("%s",s);`
- (3) `printf("the number is %d ,string is:%s", i, s);`

`printf()` 函数是一个有着变参的库函数，在 C 语言中，程序员也可以根据实际需求编写变参函数。如下程序示例代码，实现了一个变参函数 `add2()`，该函数实现多参数求和运算。

```
#include <stdio.h>
int add2(char num, ...)
{
    int sum = 0;
    int index = 0;
    int *p = NULL;
    p = (int *)&num + 1;
    for(; index < (int)num; ++index)
    {
        sum += *p++;
    }
    return sum;
}

int main()
{
    int i = 1;
    int j = 2;
    int k = 3;
    printf("%d\n",add2(3,i,j,k));
    return 0;
};
```

程序输出结果：

6

7.8.2 函数指针与指针函数有什么区别

指针函数是指带指针的函数，本质上是一个函数，函数返回类型是某一类型的指针。其形式一般如下所示：

类型标识符 *函数名(参数列表)

例如，`int *f(x,y)`，它的意思是声明一个函数 `f(x,y)`，该函数返回类型为 `int` 型指针。

而函数指针是指向函数的指针变量，即本质是一个指针变量，表示的是一个指针，它指向的是一个函数。其形式一般如下所示：

类型说明符 (*函数名)(参数)

例如, `int (*pf)(int x)`, 它的意思就是声明一个函数指针, 而 `pf=func` 则是将 `func` 函数的首地址赋值给指针。

下面为一个函数指针的实例。

```
#include <stdio.h>
#define NULL 0
#define ASGN 1
#define MUL 2
int asgn(int* a, int b)
{
    return *a = b;
}
int mul(int* a, int b)
{
    return *a * b;
}
int (*func(int op))(int*, int)
{
    switch (op)
    {
        case ASGN:
            return &asgn;
        case MUL:
            return &mul;
        default:
            return NULL;
    }
    return NULL;
}

int main( )
{
    int i = 0xFEED, j = 0xBEEF;
    printf("%x\n", func(ASGN)(&i, j));
    printf("%x\n", func(MUL)(&i, j));
    printf("%x, %x\n", i, j);
    return 0;
}
```

程序输出结果:

```
beef
8e67a321
beef, beef
```

引申: 数组指针/指针数组、函数模板/模板函数、类模板/模板类、指针常量/常量指针分别有什么区别?

(1) 数组指针/指针数组。

数组指针就是指向数组的指针, 它表示的是一个指针, 它指向的是一个数组, 它的重点是指针。例如, `int (*pa)[8]` 声明了一个指针, 该指针指向了一个有 8 个 `int` 型元素的数组。

```
#include <stdio.h>

int main( )
{
    int (*p)[4];
    int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

```

        p=&a[0];
        for(int i=0;i<12;i++)
            printf("%d ",(*p)[i]);
        printf("\n");
        return 0;
    }

```

程序输出结果:

1 2 3 4 5 6 7 8 9 10 11 12

指针数组就是指针的数组,表示的是一个数组,它包含的元素是指针,它的重点是数组。

例如, `int* ap[8]` 声明了一个数组,该数组的每一个元素都是 `int` 型的指针。

```

#include <stdio.h>

int main( )
{
    int* p[4];
    int a[4]={1,2,3,4};
    p[0]=&a[0];
    p[1]=&a[1];
    p[2]=&a[2];
    p[3]=&a[3];
    for(int i=0;i<4;i++)
        printf("%d ",*p[i]);
    printf("\n");
    return 0;
}

```

程序输出结果:

1 2 3 4

(2) 函数模板/模板函数。

函数模板是对一批模样相同的函数的说明描述,它不是某一个具体的函数;而模板函数则是将函数模板内的“数据类型参数”具体化后得到的重载函数(就是由模板而来的函数)。简单地说,函数模板是抽象的,而模板函数则是具体的。

函数模板减少了程序员输入代码的工作量,是 C++ 中功能最强的特性之一,是提高软件代码重用性的重要手段之一。函数模板的形式一般如下所示:

```

template <模板类型形参表>
<返回值类型> <函数名> (模板函数形参表)
{
    //函数体
}

```

其中<模板函数形参表>的类型可以是任何类型,包括基本数据类型和类类型。需要注意的是,函数模板并不是一个实实在在的函数,它是一组函数的描述,它并不能直接执行,需要实例化为模板函数后才能执行,而一旦数据类型形参实例化以后,就会产生一个实实在在的模板函数了。

(3) 类模板/模板类。

类模板与函数模板类似,将数据类型定义为参数,描述了代码类似的部分类的集合,具体化为模板类后,可以用于生成具体的对象。

```

template <类型参数表>
class <类名>
{
    //类说明体
};

```

```

template <类型形参表>
<返回类型><类名><类型名表>::<成员函数 1>(形参表)
{
    //成员函数定义体
}

```

其中<类型形参表>与函数模板中的<类型形参表>意义类似，而类模板本身不是一个真实的类，只是对类的一种描述，必须用类型参数将其实例化为模板类后，才能用来生成具体的对象。简而言之，类是对象的抽象，而类模板就是类的抽象。

具体而言，C++中引入模板类主要有以下 5 个方面的好处：

- 1) 可用来创建动态增长和减小的数据结构。
- 2) 它是类型无关的，因此具有很高的可复用性。
- 3) 它在编译时而不是运行时检查数据类型，保证了类型安全。
- 4) 它是平台无关的，可移植性强。
- 5) 可用于基本数据类型。

(4) 指针常量/常量指针。

指针常量是指定义的指针只能在定义的时候初始化，之后不能改变其值。其格式为：

[数据类型][*][const][指针常量名称]

例如：char* const p1; int* const p2;

const 位于指针声明符“*”的右侧，这说明声明的对象是一个常量，而对象的数据类型是指针。所以第一句定义了一个只读的字符型指针 p1；第二句定义了一个只读的整型指针 p2。常指针的值不能改变，但是其指向的内容却可以改变。

```
#include<stdio.h>
```

```

int main( )
{
    char a[5]="abcd";
    char b[5]="efgh";
    char * const p1=a;
    char * const p2=b;
    printf("Before Change:\n");
    printf("a:%s\nb:%s\n",a,b);
    *p1='1';
    b[0]='2';
    //p1=p2;
    printf("After Change:\n");
    printf("a:%s\nb:%s\n",a,b);
    return 0;
}

```

程序的输出结果如下：

```

Before Change:
a:abcd
b:efgh
After Change:
a:1bcd
b:2fgh

```

上例中，如果去掉注释行，执行 p1=p2 操作，则编译会出错：error C3892: “p1”: 不能给常量赋值(VS 2005)。指针所指向的内存地址不能更改，指针的值只能在定义的时候初始化，其他地方不能更改。

常量指针是指向常量的指针，因为常量指针指向的对象是常量，因此这个对象的值是不能够改变的。定义的格式如下：

[数据类型] [const] [*] [常量指针名称]; 或 [const] [数据类型] [*] [常量指针名称];

例如：int const *p; const int *p;

程序示例如下：

```
#include<stdio.h>

int main( )
{
    char a[5]="abcd";
    char b[5]="efgh";
    const char * p1=a;
    const char * p2=b;
    printf("Before Change:\n");
    printf("a:%s\nb:%s\np1:%s\n",a,b,p1);
    a[0]='1';
    p1=p2;
    /*p2='2';
    printf("After Change:\n");
    printf("a:%s\nb:%s\np1:%s\n",a,b,p1);
    return 0;
```

程序的输出结果：

```
Before Change:
a:abcd
b:efgh
p1:abcd
After Changed:
a:1bcd
b:efgh
p1:efgh
```

上例中，如果去掉注释行，执行*p2='2'操作，则编译会出错：error C3892: “p2”: 不能给常量赋值。

需要注意的是，指针常量强调的是指针的不可改变性，而常量指针强调的是指针对其所指对象的不可改变性，它所指向的对象的值是不能通过常量指针来改变的。对于字符串“abc”，可以这样获取其地址：&("abc");

7.8.3 C++函数传递参数的方式有哪些

当进行函数调用时，要填入与函数形式参数个数相同的实际参数，在程序运行过程中，实际参数（简称实参）就会将参数值传递给相应的形式参数（简称形参），然后在函数中实现对数据的处理和返回。C++函数传递参数的方式一般有以下4种：

（1）值传递。

当进行值传递时，就是将实参的值复制到形参中，而形参和实参不是同一个存储单元，所以函数调用结束后，实参的值不会发生改变。程序示例如下：

```
#include <iostream>
using namespace std;
void swap(int a,int b)
{
    int temp;
```

```

        temp=a;
        a=b;
        b=temp;
        cout<<a<<","<<b<<endl;
    }
    int main( )
    {
        int x=1;
        int y=2;
        swap(x,y);
        cout<<x<<","<<y<<endl;
        return 0;
    }

```

程序的输出结果:

```

2,1
1,2

```

也就是说,在进行函数调用的时候只交换了形参的值,而并未交换实参的值,形参值的改变并没有改变实参的值。

(2) 指针传递。

当进行指针传递时,形参是指针变量,实参是一个变量的地址,调用函数时,形参(指针变量)指向实参变量单元。这种方式还是“值传递”,只不过实参的值是变量的地址而已。而在函数中改变的并不是实参的值,而是实参地址所指向的变量的值。

程序示例如下:

```

#include <iostream>
using namespace std;
void swap(int *a,int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
    cout<<*a<<","<<*b<< endl;
}
int main( )
{
    int x=1;
    int y=2;
    swap(&x,&y);
    cout<<x<<","<<y<< endl;
    return 0;
}

```

程序的输出结果:

```

2,1
2,1

```

也就是说,在进行函数调用时,不仅交换了形参的值,而且交换了实参的值。

(3) 传引用。

实参地址传递到形参,使形参的地址取实参的地址,从而使形参与实参共享同一单元的方式。

程序代码示例如下:

```

#include <iostream>
using namespace std;
void swap(int &a,int &b)
{

```



```

        int temp;
        temp=a;
        a=b;
        b=temp;
        cout<<*a<<" "<<*b<<endl;
    }
    int main( )
    {
        int x=1;
        int y=2;
        swap(x,y);
        cout<<x<<" "<<y<<endl;
        return 0;
    }

```

程序的输出结果:

```

2,1
2,1

```

(4) 全局变量传递。

这里的“全局”变量并不见得就是真正的全局的，所有代码都可以直接访问的，只要这个变量的作用域足够这两个函数访问就可以了，比如一个类中的两个成员函数可以使用一个成员变量实现参数传递，或者使用 `static` 关键字定义，或者使用 `namespace` 进行限制等，而这里的成员变量在这种意义上就可以称为“全局”变量。当然，可以使用一个类外的真正的全局变量来实现参数传递，但有时并没有必要，从工程上讲，作用域越小越好。

全局变量传递的优点是效率高，但它对多线程的支持不好，如果两个进程同时调用同一个函数，而通过全局变量进行传递参数，该函数就不能总是得到想要的结果。

7.8.4 重载与覆盖有什么区别

重载是指函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。在同一可访问区域内被声明的几个具有不同参数列（参数的类型、个数、顺序不同）的同名函数，程序会根据不同的参数列来确定具体调用哪个函数。对于重载函数的调用，在编译期间就已经确定，是静态的，它们的地址在编译期间就绑定了与多态无关。注意，重载不关心函数的返回值类型。

- (1) `double calculate(double);`
- (2) `double calculate(double, double);`
- (3) `double calculate(double, int);`
- (4) `double calculate(int, double);`
- (5) `double calculate(int);`
- (6) `float calculate(float);`
- (7) `float calculate(double);`

7 个同名函数 `calculate`，(1) (2) (3) (4) (5) (6) 中任两个均构成重载，(6) 和 (7) 也能构成重载，而 (1) 和 (7) 却不能构成重载，因为 (1) 和 (7) 的参数相同。

成员函数被重载的特征如下：

- (1) 相同的范围（在同一个类中）。
- (2) 函数名字相同。
- (3) 参数不同。
- (4) `virtual` 关键字可有可无。

覆盖是指派生类中存在重新定义基类的函数，其函数名、参数列、返回值类型必须同父类中的相对应被覆盖的函数严格一致，覆盖函数和被覆盖函数只有函数体不同，当派生类对象调用子类中该同名函数时会自动调用子类中的覆盖版本，而不是父类中的被覆盖函数版本，它和多态真正相关。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态地调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的。

覆盖的特征如下：

- (1) 不同的范围（分别位于派生类与基类）。
- (2) 函数名字相同。
- (3) 参数相同。
- (4) 基类函数必须有 `virtual` 关键字。

重载与覆盖的区别如下：

- (1) 覆盖是子类 and 父类之间的关系，是垂直关系；重载是同一个类中方法之间的关系，是水平关系。
- (2) 覆盖只能由一个方法，或只能由一对方法产生关系；方法的重载是多个方法之间的关系。
- (3) 覆盖要求参数列表相同；重载要求参数列表不同。
- (4) 覆盖关系中，调用方法体是根据对象的类型（对象对应存储空间类型）来决定的，重载关系是根据调用时的实参表与形参表来选择方法体的。

程序示例如下。

```
#include <iostream>
using namespace std;

class Base
{
public:
    void f(int x)
    {
        cout << "Base::f(int) " << x << endl;
    }
    void f(float x)
    {
        cout << "Base::f(float) " << x << endl;
    }
    virtual void g(void)
    {
        cout << "Base::g(void)" << endl;
    }
};

class Derived : public Base
{
public:
    virtual void g(void)
    {
        cout << "Derived::g(void)" << endl;
    }
};

int main( )
{
    Derived d;
    Base *pb = &d;
```

```

    pb->f(42);
    pb->f(3.14f);
    pb->g( );
    return 0;
}

```

程序输出结果:

```

Base::f(int) 42
Base::f(float) 3.14
Derived::g(void)

```

上例中, 函数 `Base::f(int)` 与 `Base::f(float)` 相互重载, 而 `Base::g(void)` 被 `Derived::g(void)` 覆盖。

隐藏是指派生类的函数屏蔽了与其同名的基类函数, 规则如下:

(1) 如果派生类的函数与基类的函数同名, 但是参数不同, 则不论有无 `virtual` 关键字, 基类的函数都将被隐藏。

(2) 如果派生类的函数与基类的函数同名, 并且参数也相同, 但是基类函数没有 `virtual` 关键字, 此时基类的函数被隐藏。

在调用一个类的成员函数时, 编译器会沿着类的继承链逐级地向上查找函数的定义, 如果找到了就停止查找了。所以, 如果一个派生类和一个基类都存在同名 (暂且不论参数是否相同) 的函数, 而编译器最终选择了在派生类中的函数, 那么就说这个派生类的成员函数“隐藏”了基类的成员函数, 也就是说它阻止了编译器继续向上查找函数的定义。

回到隐藏的定义中, 前面已经说了有 `virtual` 关键字, 并且派生类函数与基类函数同名, 同参数函数构成覆盖的关系, 因此隐藏的关系只有如下的可能:

(1) 必须分别位于派生类和基类中。

(2) 必须同名。

(3) 参数不同的时候本身已经不构成覆盖关系了, 所以此时是否是 `virtual` 函数已经不重要了。

当参数相同的时候就要看是否有 `virtual` 关键字了, 有的话就是覆盖关系, 没有的时候就是隐藏关系了。

程序代码示例如下:

```

#include <iostream>
using namespace std;

class Base
{
public:
    virtual void f(float x)
    {
        cout << "Base::f(float) " << x << endl;
    }
    void g(float x)
    {
        cout << "Base::g(float) " << x << endl;
    }
    void h(float x)
    {
        cout << "Base::h(float) " << x << endl;
    }
};

class Derived : public Base

```

```

{
    public:
        virtual void f(float x)
        {
            cout << "Derived::f(float) " << x << endl;
        }
        void g(int x)
        {
            cout << "Derived::g(int) " << x << endl;
        }
        void h(float x)
        {
            cout << "Derived::h(float) " << x << endl;
        }
};

int main( )
{
    Derived d;
    Base *pb = &d;
    Derived *pd = &d;
    pb->f(3.14f);
    pd->f(3.14f);
    pb->g(3.14f);
    pd->h(3.14f);
    return 0;
}

```

程序输出结果:

```

Derived::f(float) 3.14
Derived::f(float) 3.14
Base::g(float) 3.14
Derived::h(float) 3.14

```

上例中, 函数 `Derived::f(float)` 覆盖了 `Base::f(float)`, 函数 `Derived::g(int)` 隐藏了 `Base::g(float)`, 而不是重载; 函数 `Derived::h(float)` 隐藏了 `Base::h(float)`, 而不是覆盖。

7.8.5 是否可以通过绝对内存地址进行参数赋值与函数调用

同一个数可以通过不同的方式表达出来, 对于函数的访问, 变量的赋值除了直接对变量赋值以外, 还可以通过绝对内存地址进行参数赋值与函数调用。

(1) 通过地址修改变量的值。

```

int x;
int *p;
printf("%x\n",&x);
p=(int *)0x0012ff60;
*p = 3;
printf("%d\n",x);

```

程序的输出结果:

```

12ff60
3

```

程序首先输出变量 `x` 所在地址为十六进制的 `0x12ff60` (本来应该为 8 位的十六进制数, 高位为 0 则省略掉), 然后定义一个指针变量, 让它指向该地址, 通过指针变量的值来修改变量 `x` 的值。

示例代码如下:

```
int *ptr=(int*)0xa4000000;
*ptr=0xaabb;
printf("%d\n",*ptr);
```

以上程序会崩溃，因为这样做会给一个指针分配一个随意的地址，很危险，所以这种做法是不允许的。

(2) 通过地址调用函数的执行。

```
#include <iostream>
using namespace std;

typedef void(*FuncPtr)( );

void p( )
{
    printf("MOP\n");
}

int main( )
{
    void (*ptr)( );
    p( );
    printf("%x\n",p);
    ptr = (void (*)( ))0x4110f0;
    ptr( );//函数指针执行
    ((void (*)( ))0x4110f0)( );
    ((FuncPtr)0x4110f0)( );
    return 0;
}
```

程序执行结果如下：

```
MOP
4110f0
MOP
MOP
MOP
```

首先定义一个 `ptr` 的函数指针，第一次通过函数名调用函数，输出 `Mop`，打印函数的入口地址，函数的入口地址为 `4110f0`。然后给函数指针 `ptr` 赋地址值为 `p` 的入口地址，调用 `ptr`，输出 `Mop`。接着的过程不通过函数指针直接执行，仍然使用 `p` 的入口地址调用，输出为 `MOP`。最后是通过 `typedef` 调用的直接执行。

函数名称、代码都是放在代码段的，因为是放在代码段，每次会跳到相同的地方，但参数会压栈，所以函数只根据函数名来获取入口地址，与参数和返回值无关。无论参数和返回值如何不同，函数入口地址都是一个地方。

对以下程序进行分析：

```
#include <stdio.h>

int p(int a,int b)
{
    return 3;
}

int main( )
{
    printf("%x\n",p);
    int a = p(2,3);
}
```



```

        printf("%d\n",p);
        int b = p(4,5);
        printf("%x\n",p);
        return 0;
    }

```

程序输出结果如下:

```

411159
4264281
411159

```

十六进制的 411159 转换成十进制的值为 4264281。程序中打印的 p 的入口地址,无论 p 是否调用函数,入口地址都没有改变。

分析如下代码:

```

#include <stdio.h>

int p(int a,int b)
{
    return ((a>b)?a:b);
}
int main( )
{
    int (*ptr)(int ,int);
    ptr = (int (*)(int,int))0x411159;
    int c = ptr(5,6);
    printf("%d\n",c);
    return 0;
}

```

程序输出结果:

```

6

```

通过函数指针调用有返回值和参数的函数,不使用函数名,而是用函数入口地址调用。

函数存放在内存的代码区域内,也有地址,一个函数在编译时被分配一个入口地址,将这个入口地址称为函数的指针,函数的地址就是函数的名字。函数指针不能指向不同类型或是带不同形参的函数。

7.8.6 默认构造函数是否可以调用单参数构造函数

默认构造函数不可以调用单参数的构造函数。程序示例如下:

```

#include <iostream>
using namespace std;

class A
{
public:
    A( )
    {
        A(0);
        Print( );
    }
    A(int j):i(j)
    {
        printf("Call A(int j)\n");
    }
    void Print( )
    {

```

```

        printf("Call Print( )\n");
    }
    int i;
};

int main( )
{
    A a;
    cout<<a.i<<endl;
    return 0;
}

```

程序输出结果:

```

Call A(int j)
Call Print( )
-858993460

```

以上代码希望默认构造函数调用带参构造函数，可是却未能实现。因为在默认构造函数内部调用带参的构造函数属用户行为而非编译器行为，它只执行函数调用，而不会执行其后的初始化表达式。只有在生成对象时，初始化表达式才会随相应的构造函数一起调用。

7.8.7 C++中函数调用有哪几种方式

编译器一般使用堆栈实现函数调用。堆栈是存储器的一个区域，嵌入式环境有时需要程序员自己定义一个数组作为堆栈。Windows 为每个线程自动维护一个堆栈，堆栈的大小可以设置。编译器使用堆栈来存放每个函数的参数、局部变量等信息。

由于函数调用经常会被嵌套，在同一时刻，堆栈中会存储多个函数的信息，每个函数又占用一个连续的区域，一个函数占用的区域常被称为帧（frame），编译器是从高地址开始使用堆栈的，在多线程（任务）环境，CPU 的堆栈指针指向的存储器区域就是当前使用的堆栈。切换线程的一个重要工作，就是将堆栈指针设为当前线程的堆栈栈顶地址。不同 CPU，不同编译器的堆栈布局、函数调用方法都可能不同，但堆栈的基本概念是一样的。

当一个函数被调用时，进程内核对象为其在进程的地址空间的堆栈部分分配一定的栈内存给该函数使用，函数堆栈用于：

（1）在进入函数之前，保存“返回地址”和环境变量。返回地址是指该函数结束后，从进入该函数之前的那个地址继续执行下去。

（2）在进入函数之后，保存实参或实参复制、局部变量。

函数原型：[连接规范] 函数类型 [调用约定] 函数名 参数列表 {.....}

调用约定：调用约定是决定函数实参或实参复制进入和退出函数堆栈的方式以及函数堆栈释放的方式，简单地讲就是实参或实参复制入栈、出栈、函数堆栈释放的方式。在 Win32 下有 4 种调用：

（1）_cdecl：它是 C/C++ 的默认调用方式。实参是以参数列表从右依次向左入栈，出栈相反，函数堆栈由调用方来释放，主要用在那些带有可变参数的函数上，对于传送参数的内存栈是由调用者来维护的。另外，在函数名修饰约定方面也有所不同。由于每一个调用它的函数都包含清空堆栈的代码，所以产生的可执行文件大小会比调用 _stdcall 函数的大。

（2）_stdcall：它是 WIN API 的调用约定，其实 COM 接口等只要是申明定义接口都要显示指定其调用约定为 _stdcall。实参以参数列表从右依次向左入栈，出栈相反。函数堆栈是由被调用方自己释放的。但是若函数含有可变参数，那么即使显示指定了 _stdcall，编译器也会自动把其改变成 _cdecl。

(3) `_thiscall`: 它是类的非静态成员函数默认的调用约定, 其不能用在含有可变参数的函数上, 否则编译会出错。实参以参数列表从右依次向左入栈, 出栈相反。函数堆栈是由被调用方自己释放的。但是类的非静态成员函数内部都隐含有一个 `this` 指针, 该指针不是存放在函数堆栈上, 而是直接存放在 CPU 寄存器上。

(4) `_fastcall`: 快速调用。它们的实参并不是存放在函数堆栈上, 而是直接存放在 CPU 寄存器上, 所以不存在入栈、出栈、函数堆栈释放。

需要注意的是, 全局函数或类静态成员函数, 若没指定调用, 约定默认是 `_cdecl` 或是 IDE 设置的。

7.8.8 什么是可重入函数? C 语言中如何写可重入函数

可重入函数是指能够被多个线程“同时”调用的函数, 并且能保证函数结果正确性的函数。

在 C 语言中编写可重入函数时, 尽量不要使用全局变量或静态变量, 如果使用了全局变量或静态变量, 就需要特别注意对这类变量访问的互斥。一般采用以下几种措施来保证函数的可重入性: 信号量机制、关调度机制、关中断机制等方式。

需要注意的是, 不要调用不可重入的函数, 当调用了不可重入的函数时, 会使该函数也变为不可重入的函数。一般驱动程序都是不可重入的函数, 因此在编写驱动程序时一定要注重重入的问题。

7.9 数组

数组的下标问题、越界问题一直是程序员经常忽视的问题, 但与数组相关的问题却不仅限于此: 多维数组的缺省赋值、下标越界、行存储、列存储等, 本节将详细对数组的这些问题进行逐一分析。

7.9.1 `int a[2][2]={1},{2,3}`, 则 `a[0][1]` 的值是多少

要弄清楚这个问题, 需要弄清楚二维数组默认初始化的问题。原则上来说, 二维数组在内存中既可以按行存储, 也可以按列存储, 但一般是按行存放的, 即先存储第一行的数组元素的值, 再按顺序存放第二行的数组元素的值, 以此类推。以数组 `a[m][n]` 为例, 按行存储, 在内存中的结构如下所示:

`a[0][0]→a[0][1]→a[0][2]→a[0][3]→a[1][0]→a[1][1]→a[1][2]→a[1][3]→a[2][0]→a[2][1]→...`

而对二维数组的初始化, 其初始化的形式如下:

数据类型 数组名[整常量表达式][整常量表达式]={ 初始化数据 }; 在 { } 中给出各数组元素的初值, 各初值之间用逗号分开。把 { } 中的初值依次赋给各数组元素。

二维数组的初始化一般有两种方式, 第一种方式是按行来执行 (如 `int array[2][3]={0,0,1},{1,0,0}`); 而第二种方式是把数值写在一块 (如 `int array[2][3]={0,0,1,1,0,0}`;)。

此时存在一种情况, 即只对部分元素进行初始化, 当数组中非零元素比较少时, 则可以对部分元素赋初值, 未赋值的元素自动为 0。例如, `int a[3][4]={1},{5},{9}}`, 则数组中各值可以表示如下:

```
1 0 0 0
5 0 0 0
9 0 0 0
```

再例如 `int a[4][5]={1,2},{},{0,1,3}}`, 则各值表示如下:

```

1 2 0 0 0
0 0 0 0 0
0 1 3 0 0
0 0 0 0 0

```

再例如 `int a[2][3]={{5,6},{7,8}}`，则二维数组的存储如下：

```

5 6 0
7 8 0

```

再例如 `int a[2][3]={5,6,7,8}`，则按行存储格式如下：

```

5 6 7
8 0 0

```

所以本题中，由于数组 `a` 是一个二维数组，第一行第一个数是 1，第二行第一个数是 2，第二个数是 3，其他位置数的值自动补 0，所以第一行第二个数也就是 `a[0][1]`，它的值是 0。

有一种情况比较特殊，如果对二维数组的所有元素都赋值，则数组的第一维的长度可以不指定，但第二维的长度不能省。

例如，`int m[][3]={1,2,3,4,5,6,7,8,9}`，则默认第一维的维数值为 3。

`int m[][3]={1,2,3,4,5,6,7}`，则默认第一维的维数值为 3，其中 `m[2][1]=m[2][2]=0`。

`int m[][3]={{0,0,2},{},{0,2}}`，则默认第一维的维数值为 3。

需要注意的是，只有在进行带有初始化数据的数组说明时，才允许省略第一维长度，在仅仅进行说明数组而未进行初始化数组元素时省略长度是错误的，因为编译系统无法预知数组大小，如 `int a[][4]` 就是错误的。

7.9.2 如何合法表示二维数组

对于数组 `a[3][4]`、`*(a[1]+1)`、`*(&a[1][1])`、`*(a+1)[1]` 和 `*(a+5)` 四种表示方法中，哪个不能表示 `a[1][1]`？

第一个可以，因为 `a[1]` 是第一行的地址，`a[1]+1` 偏移一个单位然后解引用取值，得到 `a[1][1]`。第二个也可以，`[]` 优先级高，`a[1][1]` 取地址再取值。第三个 `a+1` 相当于 `&a[1]`，所以 `*(a+1)=a[1]`，因此 `*(a+1)[1]=a[1][1]`。第四个 `a+5` 相当于 `&a[5]`，单从这里看就已经越界了，所以不可以。

7.9.3 a 是数组，(int*)(&a+1)表示什么意思

对于数组而言，一个数组名代表的含义是数组中第一个元素的位置，即地址，通过数组名可以访问该数组。程序示例如下：

```

#include <stdio.h>
int main()
{
    int a[5]={1,2,3,4,5};
    int b[100];
    int *ptr=(int*)(&a+1);
    printf("%d\n%d\n",*(a+1),*(ptr-1));
    printf("sizeof(b)=%d\n",sizeof(b));
    printf("sizeof(&b)=%d\n",sizeof(&b));
    return 0;
}

```

程序输出结果：

```

2
5

```

```
sizeof(b)=400
sizeof(&b)=400
```

一般而言，对指针进行加 1 操作，得到的将是下一个元素的地址，一个类型为 T 的指针移动，是以 sizeof(T) 为移动单位，如果 ptr=a+1，那么最终输出*(ptr-1)的值肯定是 2，1。而 ptr=&a+1，输出则变为 2，5。&a 是数组指针，是一个指向 int(*)[5] 的指针，但是这时候 ptr 相当于 int *[5]，也就是指向了一个含有 5 个元素的数组，这也就不难解释为什么第二个打印出来&b 的大小也是 400 了，sizeof(b)打印出来的是 b 数组的大小。sizeof(&b)同样也是。&a+1 的地址是&a 地址再加 5*sizeof(int)；它的运算单位是 int (*)[5]。

需要注意的是，数组的首地址与数组元素的首地址的问题，两者虽然值相等，但是意义却不相同，a[0]是一个元素，a 是这个数组，虽然&a[0]和&a 的值一样，前者是数组首元素的首地址，后者是数组的首地址。

ptr-1 的单位是 ptr 的类型，因此 ptr-1 的位置刚好是 a[4]，它在内存中的分布位置是和 &a+1 相邻的。但是 ptr 与(&a+1)类型是不一样的，所以 ptr-1 只会减去 sizeof(int*)。值得注意的是，a 和&a 的地址是一样的，但意思不一样，a 是数组首地址，也就是 a[0]的地址；&a 是对象（数组）首地址；a+1 是数组下一元素的地址，即 a[1]；而&a+1 是下一个对象的地址，即 a[5]。

数组下标取负值的情况

```
#include <stdio.h>
int main( )
{
    int a[5] = {0, 1, 2, 3, 4};
    int* p = &a[4];
    for (int i = -4; i <= 0; i++)
    {
        printf("%d  %d\n",p[i],&p[i]);
    }
    return 0;
}
```

程序输出结果如下：

```
0  1310572
1  1310576
2  1310580
3  1310584
4  1310588
```

从上例可以发现，在 C++ 中，数组的下标并非不可以为负数，当数组下标为负数时，编译可以通过，而且也可以得到正确的结果，只是它表示的意思却是从当前地址向前寻址，即为当前地址减去 sizeof（类型）的地址值。

7.9.4 不使用流程控制语句，如何打印出 1 ~ 1000 的整数

一般而言，采用控制流程语句，如 for、while 等都可以很容易地执行打印工作。例如，

```
int i;
for(i = 1; i <= 1000; i++)
    printf("%d\n",i);
```

但按照题目要求，不允许用流程控制语句，所以需要采取非常规的方法来完成打印工作。一般能想到的方法是采用构造函数以及宏定义两种不同的方式来实现。

方法一，采用构造函数与静态构造变量结合的方法实现。首先在类中定义一个静态成员变

量，然后在构造函数里面打印该静态变量的值，并对静态变量进行自增操作，同时主函数里面定义一个类数组，程序代码示例如下：

```
#include <stdio.h>

struct print
{
    static int a;
    print()
    {
        printf("%d\n", print::a);
        a++;
    }
};
int print::a = 1;

int main()
{
    print tt[1000];
    return 0;
}
```

方法二，可以通过使用宏定义来实现。

```
#include <stdio.h>

#define B P,P,P,P,P,P,P,P,P
#define P L,L,L,L,L,L,L,L,L,L
#define L I,I,I,I,I,I,I,I,I,N
#define I printf("%3d ",i++)
#define N printf("\n")

int main()
{
    int i = 1;
    B;
    return 0;
}
```

宏定义更简便的写法如下：

```
#include <stdio.h>
#define A(x) x;x;x;x;x;x;x;x;x;x
int main()
{
    int n = 1;
    A(A(A(printf("%d ", n++))));
    return 0;
}
```

与此题类似的题目还有：求 $1+2+\cdots+n$ ，要求不能使用乘除法、for、while、if、else、switch、case 等关键字以及条件判断语句（A?B:C）。通常针对此类题目除了用公式 $n(n+1)/2$ 之外，还可以用循环和递归两种思路，由于不能使用 for 和 while，循环已经不能再用了。同样，一般的递归函数也需要用 if 语句或者条件判断语句来判断是继续递归下去还是终止递归，所以一般的递归函数也无法满足本题的需要。

根据本题的思路可以采用构造函数与静态变量结合的方式，程序示例如下：

```
#include <iostream>
using namespace std;
```

```

class Temp
{
    public:
        Temp( );
        static void Reset( );
        static int GetSum( );
    private:
        static int N;
        static int Sum;
};

int Temp::N = 0;
int Temp::Sum = 0;

Temp::Temp( )
{
    ++ N;
    Sum += N;
}

void Temp::Reset( )
{
    N = 0;
    Sum = 0;
}

int Temp::GetSum( )
{
    return Sum;
}

int Sum(int n)
{
    Temp::Reset( );
    Temp *a = new Temp[n];
    delete []a;
    a = 0;
    return Temp::GetSum( );
}

int main( )
{
    printf("%d\n",Sum(10));
    return 0;
}

```

程序的输出结果:

55

除了以上提及的方法以外,还有很多其他方法,虽然说一般的递归函数无法解决本题问题,但是如果能够不使用条件判断语句来进行递归条件的判断,则此种递归方法可取,于是设计出了如下方法:

```

#include <stdio.h>

int func(int n)
{

```

```

    int i=1;
    (n>1)&&(i=func(n-1)+n);
    return i;
}

int main( )
{
    printf("%d\n",func(10));
    return 0;
}

```

程序输出结果:

55

该方法就非常巧妙,有兴趣的读者还可以在此基础上进行更多的发散思维。

7.9.5 char str[1024]; scanf("%s",str)是否安全

上述代码会存在安全风险,程序的潜在问题是如果用户输入了超过 1024 个长度的字符,那么就会有数组越界的问题了。

7.9.6 行存储与列存储中哪种存储效率高

有数组 a[M][N], 下面哪种算法效率更高?

(a) for(int i=0; i<M; i++)

for(int j=0; j<N; j++)

xxx=a[i][j].....

(b) for(int i=0; i<N; i++)

for(int j=0; j<M; j++)

xxx=a[j][i].....

上述两种方法中, (a) 方法的效率要高一些, C++ 采用的是行存储策略, 数组是一行一行地存的, (a) 方法是行存储, 所以效率更高。如果 C++ 是列存储的话就是后一种效率。

7.10 变量

变量是一段有名字的连续存储空间, 它是程序中数据的临时存放场所。对于计算机程序而言, 变量不是万能的, 但是没有变量却万万不能。

7.10.1 全局变量和静态变量有什么异同

全局变量的作用域是整个程序, 它只需要在一个源文件中定义, 就可以作用于所有的源文件, 其他不包含全局变量定义的源文件需要用 `extern` 关键字再次声明这个全局变量。若某一个局部重新定义了这个变量, 则全局变量作用域是除了这个局部外的整个程序, 它的生命期与程序生命期一样长。

全局变量、静态局部变量与静态全局变量都在静态存储区分配空间, 而局部变量在栈上分配空间。

静态变量存储在静态存储区, 它的生命期与程序生命期相同。例如, 某一个子程序 (子函数) 定义了一个静态变量, 当程序退出该子程序时, 这个量仍被保留, 其他非静态变量的存储单元被释放。也就是说, 非静态变量的生命期与子程序的生命期相同, 进入子程序, 分配单

元，退出则取消。下次调用子程序时非静态变量消失，静态变量却保留上次调用的结果。

总的来说，它们的相同点是都保留在静态存储区，生命期与程序生命期相同。而不同点在于全局变量具有全局作用域，静态变量具有稳健作用域。

静态变量包含静态局部变量和静态全局变量。静态局部变量具有局部作用域，只被初始化一次，自从第一次被初始化直到程序运行结束都一直存在。它和全局变量的区别在于全局变量对所有的函数都是可见的，而静态局部变量只对定义自己的函数体始终可见。静态全局变量也具有全局作用域，它与全局变量的区别在于如果程序包含多个文件的话，它作用于定义它的文件里，不能作用到其他文件里，即被 `static` 关键字修饰过的变量具有文件作用域，这样即使两个不同的源文件都定义了相同名字的静态全局变量，它们也是不同的变量。

把局部变量改变为静态变量后是改变了它的存储方式，即改变了它的生存期；把全局变量改变为静态变量后是改变了它的作用域，限制了它的适用范围。

以下程序实例是静态全局变量与静态局部变量的区别。

```
#include <stdio.h>

static int j;
int k = 0;
int m;

void fun1()
{
    static int i = 0;
    i++;
    m=i;
}

void fun2()
{
    j = 0; //如果没有此行的初始化，j 的值最后也会变为 10
    j++;
}

int main()
{
    for(k = 0 ; k < 10 ; k++)
    {
        fun1();
        fun2();
    }
    printf("%d\n",m);
    printf("%d\n",j);
    return 0;
}
```

程序的输出结果：

```
10
1
```

上例中 `i` 为静态局部变量，只被初始化一次，而 `j` 虽然也是静态变量，但在本程序实例中，它也是全局变量，所以每次函数调用的时候都会被初始化。

程序示例如下：

```
#include <stdio.h>

int my(const int a)
```

```

{
    static int count = a;
    return count + a;
}

int main( )
{
    printf("%d\n%d\n",my(4),my(5));
    return 0;
}

```

程序输出结果如下：

```

9
10

```

7.10.2 局部变量需要“避讳”全局变量吗

局部变量可以与全局变量重名，但是局部变量会屏蔽全局变量。要使用全局变量，需要使用操作符::。在函数内引用变量会用到同名的局部变量，而不会使用到全局变量，对于有些编译器来说，在同一个函数内可以定义多个同名的局部变量。例如，在两个循环体内都定义一个同名的局部变量，而那个局部变量的作用域就在那个循环体内。

具体来说，全局变量与局部变量的区别有以下4个方面的内容：

- (1) 全局变量的作用域为这个程序块，而局部变量的作用域为当前函数。
- (2) 内存存储方式不同，全局变量分配在全局数据区，后者分配在栈区。
- (3) 生命周期不同。全局变量随主程序创建而创建，随主程序销毁而销毁，局部变量在局部函数内部，甚至局部循环体等内部存在，退出就不存在了。
- (4) 使用方式不同。通过声明后全局变量程序的各个部分都可以用到，局部变量只能在局部使用。

但是需要注意的是，局部变量不可以赋值为同名全局变量。程序示例如下：

```

#include <stdio.h>

int i = 1;
int main( )
{
    int i = i;
    printf("%d\n",i);
    return 0;
}

```

程序输出结果：

```
-858993460
```

为什么输出的不是 1 而是一个随机值呢？其实上述代码合法，编译也能通过，但是不合理，`int i = i;` `i` 变量从声明的那一刻开始就是可见的了，`main()` 里的 `i` 不是 1，因为它和 `main()` 外的 `i` 无关，而是一个未定义值，所以输出就是一个随机值了。

7.10.3 如何建立和理解非常复杂的声明

- (1) 一个整型数。
- (2) 一个指向整型数的指针。
- (3) 一个指向指针的指针，它指向的指针是指向一个整型数。
- (4) 一个有 10 个整型数的数组。
- (5) 一个有 10 个指针的数组，该指针指向一个整型数。

(6) 一个指向有 10 个整型数数组的指针。

(7) 一个指向函数的指针，该函数有一个整型参数并返回一个整型数。

(8) 一个有 10 个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数。

对于上述问题，可以有如下解答：

(1) `int a;`

(2) `int *a;`

(3) `int **a;`

(4) `int a[10];`

(5) `int *a[10];`

(6) `int (*a)[10];`

(7) `int (*a)(int);`

(8) `int (*a[10])(int);`

为了更好地说明上述问题，以如下两个声明为例：

(1) `int (*a[])(int);`

(2) `int (*p[10]);`

第 (1) 种情况为数组里面是函数指针的情况，因为 `(int (*)(int))` 是一个强制转换方式，将里面的 `a[]` 这个数组转换成了一个函数指针的数组，并且该函数是一个带一个整型变量，并且返回一个整型的函数。而第 (2) 种情况是指函数返回的为指向一个一维数组的指针的情况，因为 `(int (*)[10])` 将其强制转换成了一个指针，而该指针则是一个指向一维数组的指针。

其实，在 C 语言中，每个变量声明都由两个部分组成：一个类型和一组具有特定格式的，期望用来对该类型求值的表达式。例如，`float *g(), (*h)()`，该语句表示 `*g()` 和 `(*h)()` 都是 `float` 表达式。由于 `()` 比 `*` 优先级更高，绑定得更紧密，所以 `*g()` 与 `*(g())` 表示的意义相同，即 `g` 是一个返回 `float` 指针的函数，而 `h` 是一个指向返回 `float` 的函数的指针。

对于 `float *g()` 声明，它表示 `g` 是一个返回 `float` 指针的函数，所有 `(float *)()` 就是它的模型。

除了上述提及的一些函数声明方式以外，还有一些声明方式非常复杂，如 `*(void(*)())0()`，该声明表示的意思是硬件会调用地址为 0 处的子程序，但如果写成是 `(*0)()` 并不符合要求，因为 `*` 运算符要求必须有一个指针作为它的操作数，而不能是数字。而且，这个操作数必须是一个指向函数的指针，以保证 `*` 的结果可以被调用，所以此时需要将 0 转换为一个可以描述“指向一个返回 `void` 的函数的指针”的类型，即 `(void(*)())0`。

7.10.4 变量定义与变量声明有什么区别

在 C/C++ 程序设计中，任何变量在使用前都需要进行定义或声明，定义 (definition) 为变量分配存储空间，还可以为变量指定初始值。在一个程序中，变量有且仅有一个定义。例如，`int my_array[100]`，而声明 (declaration) 是指向程序表明变量的类型和名字。定义也是声明，当定义变量时声明了它的类型和名字。可以通过使用 `extern` 关键字声明变量名而不定义它，它所说明的并非自身，而是描述其他地方创建的对象，可以多次出现，如 `extern int my_array[]`。

如果程序前面都没有出现过 `a` 这个变量，这时要使用 `a`，就必须让程序知道要使用 `a` 这个变量，这时候写入 `int a`，以前没有 `a` 这个变量的，现在程序为了记住它，就得为它分配空间，于是这是个定义。如果程序包含的其他文件里已经出现过 `a` 了，这证明程序已经为 `a` 分配内存，这时

如果要使用 `a`，只需要通过 `extern int a` 告诉程序，这个 `a` 在其他地方定义过了就可以了。

“定义也是声明”，这说明定义包括声明，对于 `int a` 来说，它既是定义又是声明，对于 `extern int a` 来说，它是声明不是定义。一般为了叙述方便，把建立存储空间的声明称定义，而不把建立存储空间的声明称为声明。

7.10.5 不使用第三方变量，如何交换两个变量的值

常用的交换两个变量的值的方法中，一般会引入第三方变量，借助它来完成。如果要交换 `int` 型的 `a` 与 `int` 型的 `b` 的值，通常的方法是：

```
int temp = a;
```

```
a = b;
```

```
b = temp;
```

可是本题却不允许使用第三方变量，于是，想到了以下几种方法来解决该问题。

(1) 算术法。

本方法利用普通的+与-运算符来实现，代码如下所示：

```
a=a+b;
```

```
b=a-b;
```

```
a=a-b;
```

以 `a=3`，`b=5` 为例，经过第一行语句的执行，`a` 的值变为了 `3+5=8`，`b` 的值不变，仍然为 `5`；经过第二行语句的执行，`a` 的值不变，为 `8`，`b` 的值为 `8-5=3`；经过第三行语句的执行，`a` 为 `8-3=5`，`b` 的值不变，为 `3`，达到了交换值的目的。

针对这种算法，是否可以将这种表达式更加精简呢？例如，通过执行语句 `a = a + b - (b=a)`，表面上看，该语句完全合法也可以达到交换的目的，而且此种方法在 `VC++6.0` 下编译可以实现交换。程序代码示例如下：

```
#include <stdio.h>
```

```
int main( )
```

```
{
```

```
    int a = 3;
```

```
    int b = 5;
```

```
    a = a + b - (b=a);
```

```
    printf("%d %d\n",a,b);
```

```
    return 0;
```

```
}
```

在 `VC` 下运行，程序输出结果：

```
5 3
```

但其实这是一种完全错误的做法，之所以 `VC++6.0` 能够运行正确，是因为这个问题是 `VC++6.0` 编译器一个自身的 `bug`。对于该语句，首先 `b` 赋值为 `a`，然后执行 `a` 的赋值语句 `a=a+b-(b=a)` 的时候，仍然减去的是 `b` 的值，而不是 `b` 赋值以后的 `a` 的值，所以 `a` 的值最终还是 `a`。所以分析得到，这种情况下，实际输出应该为 `3` 和 `3`。为了验证这一说法，在 `Visual Studio 2005` 下运行上述代码，即可得出正确的结果。

(2) 异或法。

方法如下所示：

```
a=a^b;
```

```
b=a^b;
```

```
a=a^b;
```

在分析上述代码时，首先看异或运算表（见表 7-8），它是一个有关异或运算的表格。

表 7-8 异或运算表

| | a | b |
|------------------|---|-------------------------|
| $a = a \wedge b$ | $a \wedge b$ | b |
| $b = a \wedge b$ | $a \wedge b$ | $(a \wedge b) \wedge b$ |
| $a = a \wedge b$ | $(a \wedge b) \wedge ((a \wedge b) \wedge b)$ | $(a \wedge b) \wedge b$ |

接下来把结果化简，化简结果见表 7-9 化简结果表所示。

第一步执行 $a=a\wedge b$ ，第二步执行 $b=a\wedge b$ ，将第一步带入，则根据以上步骤可以进行变换， $(a\wedge b)\wedge b=a\wedge(b\wedge b)=a\wedge 0=a$ ，可以看到，第二步完成以后 b 的值已经变成了 a 的值。第三步接着化简， $a=a\wedge b=(a\wedge b)\wedge((a\wedge b)\wedge b)=(a\wedge b)\wedge a=a\wedge a\wedge b=0\wedge b=b$ ，此时 a 也变成 b 了。通过以上 3 个步骤，最终实现了 a 与 b 的交换。

除了以上这两种方法外，也有人提出过很多其他的方法，如通过指针地址操作来实现交换的目的，但由于这些方法太过于复杂，而且不太实用，此处就不再分析了，有兴趣的读者可以搜索相关资料进行更深层次的发散思维。

7.10.6 C 与 C++变量初始化有什么不同

在 C 语言中，只能用常数对全局变量和静态变量进行初始化，否则编译器会报错。在 C 语言里，全局变量如果不初始化，默认为 0。C 语言中静态变量和全局变量（extern 外部变量属于全局变量）的分配内存空间和初始化是在编译阶段完成的，而其他变量是在编译阶段进行内存空间分配，在程序运行时执行本函数时赋予初值的。

而在 C++中，如果在一个文件中定义了 $\text{int } a = 5$ ，要在另一个文件中定义 $\text{int } b = a$ ，前面必须对 a 进行声明： $\text{extern int } a$ ，否则编译不通过，即使是这样， $\text{int } b=a$ ，这句话也是分两步进行的：在编译阶段，编译器把 b 当做是未初始化数据而将它初始化为 0；在执行阶段，在 main 被执行前有一个全局对象的构造过程， $\text{int } b = a$ ，被当做是 int 型对象 b 的复制初始化构造来执行。

在 C++中全局对象、变量的初始化是独立的，如果不是像 $\text{int } a=5$ 这样的已初始化数据，那么就是像 b 这样的未初始化数据。而 C++中全局对象、变量的构造函数调用顺序是跟声明有一定关系的，即在同一个文件中先声明的先调用。对于不同文件中的全局对象、变量，它们的构造函数调用顺序是未定义的，取决于具体的编译器，不同的编译器、连接器，结果都可能不同，另外，连接时，指定.obj 文件的顺序也有关系。

引申：C 语言中各种变量的默认初始值是什么？

全局变量放在内存的全局数据区，由编译器建立，如果在定义的时候不做初始化，则系统将自动为其初始化，数值型为 0，字符型为 NULL，即 0，指针变量也被赋值为 NULL。静态变量的情况与全局变量类似。而非静态局部变量如果不显示初始化，那么其内容是不可预料的，将是随机数，会很危险，对系统的安全造成非常大的隐患。

7.11 字符串

字符串是编程语言的基础，对于字符串的处理一直是程序设计的重要组成部分，所以在面

试笔试中，经常会考查字符串处理函数的机理。掌握常见的字符串处理函数原理，并能自己实现其原型，对于应对程序员面试笔试非常重要。

7.11.1 不使用 C/C++ 字符串库函数，如何自行编写 strcpy() 函数

strcpy 的原型为 `extern char *strcpy(char *dest, const char *src);` 它包含在头文件 `string.h` 中，它的返回指向 `dest` 的指针，其功能是把 `src` 所指由 `NULL` 结束的字符串复制到 `dest` 所指的数组中。值得注意的是，`src` 和 `dest` 所指内存区域不可以重叠，且 `dest` 必须有足够的空间来容纳 `src` 的字符串，`src` 字符串尾的字符串结束标识符 `'\0'` 也会被复制过去。

```
char * strcpy(char *strDest, const char *strSrc);
{
    assert((strDest!=NULL) && (strSrc !=NULL));
    char *address = strDest;
    while( (*strDest++ = * strSrc++) != '\0')
        ;
    return address ;
}
```

C 语言的 `assert()` 在 `<assert.h>` 中，当使用 `assert` 时，给它一个参数，即一个判断为真的表达式。`strcpy` 能把 `strSrc` 的内容复制到 `strDest`，返回类型为 `char *` 主要是为了实现链式表达式。例如：

```
int length=strlen(strcpy(strDest, "hello world"));
strcpy(strDest, strcpy(strDest1, strSrc));
```

可以将 `strSrc` 复制到 `strDest1` 与 `strDest` 中，也就是说，可以将函数的返回值做为另一个函数的参数。

程序代码示例如下：

```
#include <stdio.h>
#include <string.h>

int main( )
{
    char str[] = "hello world";
    strcpy(str, "h");
    printf("%s\n", str);
    printf("%c\n", str[3]);
    return 0;
}
```

程序输出结果：

```
h
l
```

因为 `strcpy` 将 `hi` 以及 `\0` 一直复制到 `str` 的地址空间内，覆盖了 `hello world` 字符串（`hello world` 是在栈区上分配的内存空间），但是后续的字符并没有被替换，如 `str[3]` 仍然是 `l`。将 `char str[] = "hello world";` 改为 `char* str = "hello world";` 是不允许的，因为 `str` 此时是一个指针变量。

再例如：

```
#include <stdio.h>
#include <string.h>

int main( )
{
    char s[] = "123456789";
    char d[] = "123";
```

```

    strcpy(d,s);
    printf("%s\n%s\n",d,s);
    return 0;
}

```

程序输出结果:

```

123456789
56789

```

上例中, 数组 s 和数组 d 在内存中的存储结构如图 7-5 所示。

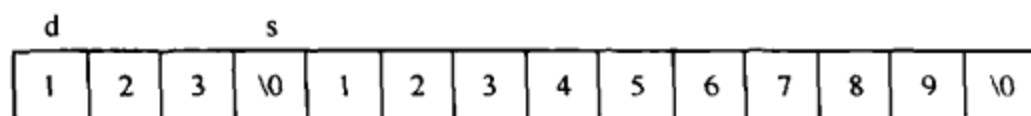


图 7-5 数组 s 和数组 d 在内存中的存储结构

strcpy 的功能就是把“源字符串”s 复制到“目的字符串”d, 直到遇到“源字符串”的结束标识符'\0', 复制后的数组 d 和数组 s 的存储结构如图 7-6 所示。

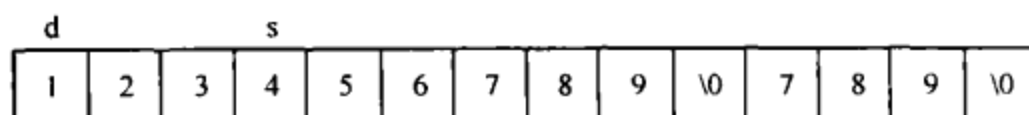


图 7-6 复制后的数组 s 和数组 d 的存储结构

所以 d 变为“123456789”, s 变为“56789”。

程序代码示例如下:

```

void test()
{
    char string[10];
    char* str1 = "0123456789";
    strcpy( string, str1 );
}

```

上例中, 字符串 str1 需要 11 个字节才能放下 (包括末尾的'\0'), 而 string 只有 10 个字节的空间, 所以执行 strcpy 会导致数组越界。

程序代码示例如下:

```

void test()
{
    char string[10], str1[10];
    int i;
    for(i=0; i<10; i++)
        str1[i] = 'a';
    strcpy( string, str1 );
}

```

上例中字符数组 str1 没有结尾符'\0', 不能在数组内结束, 执行 strcpy(string, str1)语句, 会使从 str1 内存起复制到 string 内存的字节数具有不确定性。

程序代码示例如下:

```

void test(char* str1)
{
    char string[10];
    if(strlen( str1 )<=10)
        strcpy(string, str1);
}

```

上例中 if(strlen(str1) <= 10)应改为 if(strlen(str1) < 10), 因为 strlen 的结果未统计'\0'所占用的 1 个字节。

7.11.2 如何把数字转换成字符串

C 语言中常用到字符串与数字之间的相互转换,常见的此类库函数有 `atof` (字符串转换成浮点数)、`atoi` (字符串转换成整型数)、`atol` (字符串转换成长整型数)、`itoa` (整型数转换成字符串)、`ltoa` (长整型数转换为字符串) 等。

为了考查求职者对基本功的掌握,在程序员的面试笔试中仍然经常会见到此类题目,让求职者自定义此类函数的实现,此类题目虽然难度不大,但是需要认真仔细对待。

以自定义 `Myatoi()` 与 `Myitoa()` 函数为例,分别实现自定义字符串转换为整型数函数与自定义整型数转换为字符串函数。以下为自定义 `Myatoi()` 函数的实现以及测试代码。

```
#include<stdio.h>

int Myatoi(char* str)
{
    if (str == NULL)
    {
        printf("Invalid Input");
        return -1;
    }
    while(*str == ' ' || *str == '\t')
    {
        str++;
    }
    int nSign = (*str == '-') ? -1 : 1; //确定符号位
    if(*str == '+' || *str == '-')
    {
        *str++;
    }
    int nResult = 0;
    while(*str >= '0' && *str <= '9')
    {
        nResult = nResult * 10 + (*str - '0');
        *str++;
    }
    return nResult * nSign;
}

int main( )
{
    printf("%d\n",Myatoi("12345"));
    return 0;
}
```

程序输出结果

12345

以下为自定义 `Myitoa` 函数的实现以及测试代码。

```
#include <stdio.h>

char* Myitoa(int num)
{
    char str[1024];
    int sign = num,i = 0,j = 0;
    char temp[11];
    if(sign<0)
    {
```

```

        num = -num;
    };
    do
    {
        temp[i] = num%10+'0';
        num/=10;
        i++;
    }while(num>0);
    if(sign<0)
    {
        temp[i++] = '-';
    }
    temp[i] = '\0';
    i--;
    while(i>=0)
    {
        str[j] = temp[i];
        j++;
        i--;
    }
    str[j] = '\0';
    return str;
}

int main( )
{
    printf("%s\n",Myitoa(-12345));
    return 0;
}

```

程序输出结果:

12345

7.11.3 如何自定义内存复制函数 memcpy()

memcpy 是 C 语言中的内存复制函数，它的函数原型为 `void *memcpy(void *dest, const void *src, size_t n)`。它的目的是将 src 指向地址为起始地址的连续 n 个字节的数据复制到以 dest 指向地址为起始地址的空间内，函数返回指向 destin 的指针。需要注意的是，src 和 dest 所指内存区域不能重叠，同时，与 strcpy 相比，memcpy 遇到 '\0' 不结束，而是一定会复制完 n 个字节。而且如果目标数组 dest 本身已有数据，执行 memcpy() 之后，将覆盖原有数据（最多覆盖 n）。如果要追加数据，则每次执行 memcpy 后，要将目标数组地址增加到要追加数据的地址。

memcpy() 函数用来做内存复制，可以拿它来复制任何数据类型的对象，可以指定复制的数据长度，例如：

```

char a[100],b[50];
memcpy(b,a,sizeof(a));

```

strcpy 和 memcpy 都是用于从一块内存复制一段连续的数据到另一块内存，区别是终结标识不同。strcpy(a, b) 从 b 复制内容到 a，然后从 b+1 复制内容到 a+1，依次类推，直到 b+i 的内容是 '\0'。而 memcpy(a, b, c) 从 b 开始复制 c 字节内容到 a，相比 strcpy，memcpy 是确定复制 c 个字节的。所以只要保证 b 开始有 c 字节有效数据，a 开始有 c 字节内存空间就行。

自定义内存复制构造函数示例如下：

```
#include <stdio.h>
```

```
void* MyMemCpy(void *dest, const void *src, size_t count)
```

```

{
    char* pdest = static_cast<char*>( dest );
    const char* psrc = static_cast<const char*>(src);
    if( (pdest>psrc) && (pdest<(psrc+count)))
    {
        for( size_t i=count-1; i!=-1; --i )
            pdest[i] = psrc[i];
    }
    else
    {
        for( size_t i=0; i<count; ++i )
            pdest[i] = psrc[i];
    }
    return dest;
}

int main( )
{
    char str[] = "0123456789";
    MyMemCpy(str+1, str+0, 9);
    printf("%s\n",str);
    MyMemCpy(str, str+5, 5 );
    printf("%s\n",str);
    return 0;
}

```

程序输出结果:

```

0012345678
4567845678

```

7.12 编译

编译是程序执行过程中的一个重要步骤，了解程序的编译过程以及编译方式都对程序的开发起着至关重要的作用。通过对编译过程的理解，有助于程序员编写逻辑清晰、高效的代码，有助于减少程序中存在的潜在 Bug。

7.12.1 编译和链接的区别是什么

在多道程序环境中，要想将用户源代码变成一个可以在内存中执行的程序，通常分为 3 个步骤：编译、链接、载入。

(1) 编译：将预处理生成的文件，经过词法分析、语法分析、语义分析以及优化后编译成若干个目标模块。可以理解为将高级语言翻译为计算机可以理解的二进制代码，即机器语言。

(2) 链接：由链接程序将编译后形成的一组目标模块以及它们所需要的库函数链接在一起，形成一个完整的载入模型。链接主要解决模块间的相互引用问题，分为地址和空间分配，符号解析和重定位几个步骤。在编译阶段生成目标文件时，会暂时搁置那些外部引用，而这些外部引用就是在链接时进行确定的，链接器在链接时，会根据符号名称去相应模块中寻找对应符号。待符号确定之后，链接器会重写之前那些未确定的符号的地址，这个过程就是重定位。链接一般分为静态链接、载入时动态链接以及运行时动态链接 3 种。

(3) 载入：由载入程序将载入模块载入内存。

编译和链接是为将用户程序从硬盘上调入内存并将其转换成可执行程序服务的。用编译器

时的 compile 就是在进行编译, link 就是链接, 运行程序可以看到。

以 C/C++ 语言为例, 把源文件编译成中间代码文件, 在 Windows 下面为 .obj 文件, 在 UNIX、Linux 下面就是 .o 文件, 即 Object File, 该动作被称为编译。然后再把大量的 Object File 合成执行文件, 这个动作称为链接。

编译时, 编译器需要的是语法正确, 函数与变量的声明正确。而一般来说, 每个源文件都应该对应于一个中间目标文件 (.o 文件或是 .obj 文件)。链接时, 主要是链接函数和全局变量, 所以可以使用这些中间目标文件 (.o 文件或是 .obj 文件) 来链接应用程序。链接就是那些目标文件之间相互链接自己所需要的函数和全局变量, 而函数可能来源于其他目标文件或库文件。

7.12.2 编译型语言与解释型语言的区别是什么

编译型语言: 编译是指在应用源程序执行之前, 就将程序源代码“翻译”成目标代码(机器语言), 因此其目标程序可以脱离其语言环境独立执行, 使用比较方便、效率较高。但应用程序一旦需要修改, 必须先修改源代码, 再重新编译生成新的目标文件 (*.obj) 才能执行, 只有目标文件而没有源代码, 修改很不方便。现在大多数的编程语言都是编译型的。编译程序将源程序翻译成目标程序后保存在另一个文件中, 该目标程序可脱离编译程序直接在计算机上多次运行。大多数软件产品都是以目标程序形式发行给用户的, 不仅便于直接运行, 同时又使他人难于盗用其中的技术, C、C++、Fortran、Visual Foxpro、Pascal、Delphi、Ada 都是编译实现的。

解释型语言: 解释型语言的实现中, 翻译器并不产生目标机器代码, 而是产生易于执行的中间代码。这种中间代码与机器代码是不同的, 中间代码的解释是由软件支持的, 不能直接使用硬件, 软件解释器通常会导致执行效率较低。用解释型语言编写的程序是由另一个可以理解中间代码的解释程序执行的。与编译程序不同的是, 解释程序的任务是逐一将源程序的语句解释成可执行的机器指令, 不需要将源程序翻译成目标代码后再执行。解释程序的优点是当语句出现语法错误时, 可以立即引起程序员的注意, 而程序员在程序开发期间就能进行校正。对于解释型 Basic 语言, 需要一个专门的解释器解释执行 Basic 程序, 每条语句只有在执行时才被翻译。这种解释型语言每执行一次就翻译一次, 因而效率低下。一般地, 动态语言都是解释型的, 例如 Tcl、Perl、Ruby、VBScript、JavaScript 等。

需要注意的是, Java 是一类特殊的编程语言, Java 程序也需要编译, 但是却没有直接编译为机器语言, 而是编译为字节码, 然后在 Java 虚拟机上以解释方式执行字节码。

7.12.3 如何判断一段程序是由 C 编译程序还是由 C++ 编译程序编译的

如果编译器在编译 cpp 文件, 那么 _cplusplus 就会被定义, 如果是一个 C 文件在被编译, 那么 __STDC__ 就会被定义。__STDC__ 是预定义宏, 当它被定义后, 编译器将按照 ANSI C 标准来编译 C 语言程序。

所以, 可以采用如下程序示例判断。

```
#include<stdio.h>

#ifdef _cplusplus
#define USING_C 0
#else
#define USING_C 1
#endif
```

```
#include <stdio.h>

int main( )
{
    if(USING_C)
        printf("C\n");
    else
        printf("C++\n");
    return 0;
}
```

在 C++ 编译环境下，程序输出结果如下：

C++

编写 C 与 C++ 兼容的代码所需的宏如下：

```
#ifdef _cplusplus
extern "C" {
#endif
    // 具体的代码
#ifdef _cplusplus
}
#endif
```

在上例中，`_cplusplus` 是 `cpp` 中的自定义宏，当定义了这个宏时，其表示的意思为这是一段 `cpp` 的代码。也就是说，上面的代码的含义是如果这是一段 `cpp` 的代码，那么加入 `extern "C">{和}` 处理其中的代码。

考虑到 C 语言没有重载函数的概念，所以 C 编译器编译的程序里，所有函数只有函数名对应的入口。而由于 C++ 支持函数重载，如果只有函数名对应的入口，则会出现混淆，所以 C++ 编译器编译的程序，应该是函数名+参数类型列表对应到入口。

因为 `main` 函数是整个程序的入口，所以 `main` 是不能有重载的。如果一个程序只有 `main()` 函数，是无法确认是 C 还是 C++ 编译器编译的，此时可以通过 `nm` 来查看函数名入口。例如，函数 `int foo(int i, float j)`，C 编译的程序通过 `nm` 查看如下：`foo 0x567xxxxxx` (地址)。C++ 编译程序，通过 `nm` 查看为 `foo(int, float) 0x567xxxxxx`。

需要注意的是，如果要在 C++ 编译器里使用通过 C 编译的目标文件，必须通知 C++ 编译器。

7.12.4 在 C++ 程序中调用被 C 编译器编译后的函数，为什么要加

extern "C"

C++ 语言是一种面向对象编程语言，支持函数重载，而 C 语言是面向过程的编程语言，不支持函数重载，所以函数被 C++ 编译后在库中的名字与 C 语言的不同。如果声明一个 C 语言函数 `float f(int a, char b)`，C++ 的编译器就会将这个名称变成像 `_f_int_char` 之类的东西以支持函数重载。然而 C 语言编译器的库一般不执行该转换，所以它的内部名为 `_f`，这样连接器将无法解释 C++ 对函数 `f()` 的调用。

C++ 提供了 C 语言 Alternate linkage specifications (替代连接说明) 符号 `extern "C"` 来解决名字匹配问题，`extern` 是 C/C++ 语言中表明函数和全局变量作用范围 (可见性) 的关键字，该关键字告诉编译器，其声明的函数和变量可以在模块或其他模块中使用。`extern` 后跟一个字符串来指定想声明的函数的连接类型，后面是函数声明。

```
extern "C" float f(int a, char b);
```

该语句的目的是告诉编译器 `f()` 是 C 连接的，这样 C++ 就不会转换函数名。标准的连接类

型指定符有“C”和“C++”两种，但编译器开发商可以选择用同样的方法支持其他语言。如果有一组替代连接的声明，可以把它们放在花括号里：

```
extern "C"
{
    float f(int a, char b);
    ...//其他函数
}
```

或者写成

```
extern "C"
{
    #include "Myheader.h"
    ...//其他 C 头文件
}
```

这就告诉 C++编译器，函数 f 是 C 连接的，应该到库中找名字_f 而不是找_f_int_char。C++编译器开发商已经对 C 标准库的头文件作了 extern“C”处理，所以可以用#include 直接引用这些头文件。

7.12.5 两段代码共存于一个文件，编译时有选择地编译其中的一部分，如何实现

可以通过以下两种方法实现：

- (1) 在源码中使用条件编译语句，然后在程序文件中定义宏的形式来选择需要的编译代码。
- (2) 在源码中使用条件编译语句，然后在编译命令的命令中加入宏定义命令来实现选择编译。

7.13 面向对象相关

面向对象思想是程序设计历史上一次伟大的创新，面向对象的提出极大地提高了程序设计的效率，为程序设计的重用性奠定了坚实的基础，面向对象思想已经广泛应用在现今主流的编程语言中，如 C++、Java、C#等。

7.13.1 面向对象与面向过程有什么区别

面向对象是当今软件开发方法的主流方法之一，它是把数据及对数据的操作方法放在一起，作为一个相互依存的整体，即对象。对同类对象抽象出其共性，即类，类中的大多数数据，只能被本类的方法进行处理。类通过一个简单的外部接口与外界发生关系，对象与对象之间通过消息进行通信。程序流程由用户在使用中决定。例如，站在抽象的角度，人类具有身高、体重、年龄、血型等一些特称。人类仅仅只是一个抽象的概念，它是不存在的实体，但是所有具备人类这个群体的属性与方法的对象都叫人，这个对象人是实际存在的实体，每个人都是人这个群体的一个对象。

而面向过程是一种以事件为中心的开发方法，就是自顶向下顺序执行，逐步求精，其程序结构是按功能划分为若干个基本模块，这些模块形成一个树状结构，各模块之间的关系也比较简单，在功能上相对独立，每一模块内部一般都是由顺序、选择和循环三种基本结构组成的，其模块化实现的具体方法是使用子程序，而程序流程在写程序时就已经决定。例如五子棋，面向过程的设计思路就是首先分析问题的步骤：第一步，开始游戏；第二步，黑子先走；第三步，绘制画面；第四步，判断输赢；第五步，轮到白子；第六步，绘制画面；第七步，判断输赢；第八步，返回步骤 2；第九步，输出最后结果。把上面每个步骤用分别的函数来实现，就是一个面向过程的开发方法。

具体而言,两者主要有以下几个方面的不同之处:

(1) 出发点不同。

面向对象是用符合常规思维方式来处理客观世界的问题,强调把问题域的要领直接映射到对象及对象之间的接口上。而面向过程方法则不然,它强调的是过程的抽象化与模块化,它是以过程为中心构造或处理客观世界问题的。

(2) 层次逻辑关系不同。

面向对象方法则是用计算机逻辑来模拟客观世界中的物理存在,以对象的集合类作为处理问题的基本单位,尽可能地使计算机世界向客观世界靠拢,以使问题的处理更清晰直接。面向对象方法是用类的层次结构来体现类之间的继承和发展。而面向过程方法处理问题的基本单位是能清晰准确地表达过程的模块,用模块的层次结构概括模块或模块间的关系与功能,把客观世界的问题抽象成计算机可以处理的过程。

(3) 数据处理方式与控制程序方式不同。

面向对象方法将数据与对应的代码封装成一个整体,原则上其他对象不能直接修改其数据,即对象的修改只能由自身的成员函数完成。控制程序方式上是通过“事件驱动”来激活和运行程序。而面向过程方法是直接通过程序来处理数据,处理完毕后即可显示处理结果。在控制程序方式上是按照设计调用或返回程序,不能自由导航,各模块之间存在着控制与被控制、调用与被调用的关系。

(4) 分析设计与编码转换方式不同。

面向对象方法贯穿软件生命周期的分析、设计及编码之间,是一种平滑过程,从分析到设计再到编码采用一致性的模型表示,即实现的是一种无缝连接。而面向过程方法强调分析、设计及编码之间按规则进行转换,贯穿软件生命周期的分析、设计及编码之间,实现的是一种有缝的连接。

7.13.2 面向对象的基本特征有哪些

面向对象方法首先对需求进行合理分层,然后构建相对独立的业务模块,最后通过整合各模块,达到高内聚、低耦合的效果,从而满足客户要求。具体而言,它有3个基本特征:封装、继承和多态。

(1) 封装是指将客观事物抽象成类,每个类对自身的数据和方法实行保护。类可以把自己的数据和方法只让可信的类或者对象操作,对不可信的进行信息隐藏。C++中类是一种封装手段,采用类来描述客观事物的过程就是封装,本质上是对客观事物的抽象。

(2) 继承可以使用现有类的所有功能,而不需要重新编写原来的类,它的目的是为了进行代码复用和支持多态。它一般有3种形式:实现继承、可视继承、接口继承。其中,实现继承是指使用基类的属性和方法而无需额外编码的能力;可视继承是指子窗体使用父窗体的外观和实现代码;接口继承仅使用属性和方法,实现滞后到子类实现。前两种(类继承)和后一种(对象组合=>接口继承以及纯虚函数)构成了功能复用的两种方式。通过继承创建的新类称为“派生类”或“子类”,被继承的类称为“父类”、“基类”或“超类”,而继承的过程就是从一般到特殊(具体)的过程。

(3) 多态是指同一个实体同时具有多种形式,它主要体现在类的继承体系中,它是将父对象设置成为和一个或更多的它的子对象相等的技术,赋值以后,父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单地说,就是允许将子类类型的指针赋值给父类类型的指针。编译时多态是静态多态,在编译时就可以确定对象使用的形式。

7.13.3 什么是深复制？什么是浅复制

如果一个类拥有资源（堆或者是其他系统资源），当这个类的对象发生复制过程时，资源重新分配，这个过程就是深复制；反之对象存在资源，但复制过程并未复制资源的情况视为浅复制。

例如，在某些状况下，类内成员变量需要动态开辟堆内存，如果实行位复制，也就是把对象里的值完全复制给另一个对象，如 $A=B$ ，这时，如果类 B 中有一个成员变量指针已经申请了内存，那么类 A 中的那个成员变量也指向同一块内存。这就出现了问题：当 B 把内存释放了，如通过析构函数，这时 A 内的指针就变成野指针了，导致运行错误。

深复制的程序示例如下：

```
#include <iostream>
using namespace std;
class CA
{
public:
    CA(int b,char* cstr);
    CA(const CA& C);
    void Show( );
    ~CA( );
private:
    int a;
    char *str;
};

CA::CA(int b,char* cstr)
{
    a=b;
    str=new char[b];
    strcpy(str,cstr);
}

CA::CA(const CA& C)
{
    a=C.a;
    str=new char[a]; //深复制
    if(str!=0)
        strcpy(str,C.str);
}

void CA::Show( )
{
    cout<<str<<endl;
}

CA::~~CA( )
{
    delete str;
}

int main( )
{
    CA A(10,"Hello");
    CA B=A;
```

```

        B.Show( );
        return 0;
    }

```

程序输出结果:

Hello

浅复制资源后,在释放资源时会产生资源归属不清的情况,导致程序运行出错。`Test(Test &c_t)`是自定义的复制构造函数,复制构造函数的名称必须与类名称一致,函数的形式参数是本类型的一个引用变量,且必须是引用。当用一个已经初始化过的自定义类类型对象去初始化另一个新构造的对象时,复制构造函数就会被自动调用。如果没有自定义复制构造函数时,系统将会提供给一个默认的复制构造函数来完成这个过程,上面代码的复制核心语句就是通过 `Test(Test &c_t)` 复制构造函数内的 `p1=c_t.p1`, 语句完成的。

7.13.4 什么是友元

类具有封装、继承、多态、信息隐藏的特性,只有类的成员函数才可以访问类的标记为 `private` 的私有成员,非成员函数可以访问类中的公有成员,但是却无法访问私有成员,为了使非成员函数可以访问类的成员,唯一的做法就是将成员都定义为 `public`,但如果将数据成员都定义为公有的,这又破坏了信息隐藏的特性。而且,对某些成员函数多次调用时,由于参数传递,类型检查 and 安全性检查等都需要时间开销,从而影响程序的运行效率。

友元正好解决了这一棘手的问题。在使用友元函数时,一般需要注意以下几个方面的问题:

(1) 必须在类的说明中说明友元函数,说明时以关键字 `friend` 开头,后跟友元函数的函数原型,友元函数的说明可以出现在类的任何地方,包括 `private` 和 `public` 部分。

(2) 友元函数不是类的成员函数,所以友元函数的实现与普通函数一样,在实现时不用“`::`”指示属于哪个类,只有成员函数才使用“`::`”作用域符号。

(3) 友元函数不能直接访问类的成员,只能访问对象成员。

(4) 友元函数可以访问对象的私有成员,但普通函数不行。

(5) 调用友元函数时,在实际参数中需要指出要访问的对象。

(6) 类与类之间的友元关系不能继承。

友元一般定义在类的外部,但它需要在类体内进行说明,为了与该类的成员函数加以区别,在说明时前面加以关键字 `friend`。需要注意的是,友元函数不是成员函数,但是它可以访问类中的私有成员。友元的作用在于提高程序的运行效率,但是它破坏了类的封装性和隐藏性,使得非成员函数可以访问类的私有成员。

友元可以是一个函数,该函数被称为友元函数;友元也可以是一个类,该类被称为友元类。友元函数是指某些虽然不是类成员却能够访问类的所有成员的函数。友元函数的特点是能够访问类中的私有成员的非成员函数。从语法上看,友元函数与普通函数一样,即在定义上和调用上与普通函数一样。成员函数和非成员函数最大的区别在于成员函数可以是虚的,而非成员函数不行。所以,如果有一个函数必须进行动态绑定,就要采用虚函数,而虚函数必定是某个类的成员函数。

如下为一个友元函数的例子。

```

#include <iostream>
#include <string>
using namespace std;
class Fruit

```

```

    {
        public:
            Fruit(const string &nst = "apple",const string &cst = "green"):name(nst),colour(cst)
            {
            }
            ~Fruit( )
            {
            }
            friend istream& operator>>(istream&,Fruit&);
            friend ostream& operator<<(ostream&,const Fruit&);
            void print( )
            {
                cout<<colour<<" "<<name<<endl;
            }
            string name;
            string colour;
    };

    ostream& operator<<(ostream &out,const Fruit &s) //重载输出操作符
    {
        out<<s.colour<<" "<<s.name;
        return out;
    }

    istream& operator>>(istream& in,Fruit &s) //重载输入操作符
    {
        in>>s.colour>>s.name;
        if(!in)
            cerr<<"Wrong input!"<<endl;
        return in;
    }

    int main( )
    {
        Fruit apple;
        cin>>apple;
        cout<<apple;
        return 0;
    }

```

注意：由于 VC++6.0 自身的 Bug，以上程序在 VC++6.0 编译无法通过，需要在 Visual Studio 2005 及以上版本下编译才能通过。

7.13.5 复制构造函数与赋值运算符的区别是什么

复制构造函数是一种特殊的构造函数，在生成一个实例的时候，一般会同时生成一个默认的复制构造函数，复制构造函数完成一些基于同一类的其他对象的构建及初始化工作。具体而言，拷贝构造函数有如下特点：

- (1) 该函数名与类同名，因为它也是一种构造函数，并且该函数不指定返回类型。
- (2) 该函数只有一个参数，并且是对某个对象的引用。
- (3) 每个类都必须有一个复制构造函数。

(4) 如果程序员没有显式地定义一个复制构造函数，那么，C++编译器会自动生成一个缺省的拷贝构造函数。

(5) 复制构造函数的目的是建立一个新的对象实体，所以一定要保证新创建的对象有独立的内存空间，而不是与先前的对象共用。

虽然说在生成实例时会同时生成一个默认的复制构造函数，但是在定义一些类时，有时需要甚至强烈推荐显式地定义复制构造函数用来实现特定的用户操作。

而赋值操作符则不一样，它用已存在的对象来创建另一个对象，给对象赋予一个新的值，由于赋予的是新值，反过来说，该对象原来就有值，所以赋值函数只能被已经存在了的对象调用，而不能凭空产生。而且如果不主动编写拷贝构造函数和赋值函数，编译器将以“位复制”的方式自动生成默认的函数，如果类中含有指针变量，那么这两个默认的函数就隐含了错误。

具体而言，复制构造函数相比较赋值运算符有以下3个方面的不同：

(1) 复制构造函数生成新的类对象，而赋值运算符不能。

(2) 由于复制构造函数是直接构造一个新的类对象，所以在初始化这个对象之前不用检验源对象是否和新建对象相同。而赋值运算符则需要这个操作，另外赋值运算中如果原来的对象中有内存分配，要先把内存释放掉。

(3) 当类中有指针类型的成员变量时，一定要重写复制构造函数和赋值构造函数，不能使用默认的。

简单点说，当进行一个类的实例初始化时，也就是构造时，调用的是构造函数，但如是用其他实例来初始化，则调用复制构造函数，非初始化时对这个实例进行赋值调用的是赋值运算符。

程序示例如下：

```
#include <iostream>
using namespace std;

class CTest
{
public:
    int m_muber;
    CTest():m_muber(0)
    {
        cout << "CTest()" << endl;
    }
    CTest(const CTest& t)
    {
        cout << "CTest(const CTest& t)" << endl;
        this->m_muber = t.m_muber;
    }
    CTest(const int& t)
    {
        cout << "CTest(const int& t)" << endl;
        this->m_muber = t;
    }
    CTest& operator=(const CTest& t)
    {
        cout << "CTest& operator=(const CTest& t)" << endl;
        this->m_muber = t.m_muber;
        return *this;
    }
    CTest& operator=(const int& t)
    {
        cout << "CTest& operator=(const int& t)" << endl;
        this->m_muber = t;
        return *this;
    }
}
```

```
};

int main( )
{
    cout << "CTest a: ";
    CTest a;
    cout << "CTest b(a) : ";
    CTest b(a);
    cout << "CTest c = a : ";
    CTest c = a;
    cout << "CTest d = 5 : ";
    CTest d = 5;
    cout << "b = a : ";
    b = a;
    cout << "c = 5 : ";
    c = 5;
    return 0;
}
```

程序输出结果:

```
CTest a: CTest( )
CTest b(a) : CTest(const CTest& t)
CTest c = a : CTest(const CTest& t)
CTest d = 5 : CTest(const int& t)
b = a : CTest& operator=(const CTest& t)
c = 5 : CTest& operator=(const int& t)
```

7.13.6 基类的构造函数/析构函数是否不能被派生类继承

基类的构造函数/析构函数不能被派生类继承。

基类的构造函数不能被派生类继承，派生类中需要声明自己的构造函数。在设计派生类的构造函数时，不仅要考虑派生类所增加的数据成员初始化，也要考虑基类的数据成员的初始化。声明构造函数时，只需要对本类中新增成员进行初始化，对继承来的基类成员的初始化，需要调用基类构造函数完成。

基类的析构函数也不能被派生类继承，派生类需要自行声明析构函数。声明方法与一般（无继承关系时）类的析构函数相同，不需要显式地调用基类的析构函数，系统会自动隐式调用。需要注意的是，析构函数的调用次序与构造函数相反。

7.13.7 初始化列表和构造函数初始化的区别是什么

初始化列表一般如下所示。

```
Object::Object(int _x, int _y) :x(_x),y(_y){}
```

构造函数初始化一般通过构造函数实现，示例如下。

```
Object::Object(int _x, int _y)
{
    x=_x;
    y=_y;
}
```

上面的构造函数（使用初始化列表的构造函数）显式地初始化类的成员；而没使用初始化列表的构造函数是对类的成员赋值，并没有进行显式的初始化。

初始化和赋值对内置类型的成员没有什么大的区别，在成员初始化列表和构造函数体内进

行,在性能和结果上都是一样的。对非内置类型成员变量,因为类类型的数据成员对象在进入函数体前已经构造完成,也就是说在成员初始化列表处进行构造对象的工作,调用构造函数,在进入函数体之后,进行的是对已经构造好的类对象的赋值,又调用一个复制赋值操作符才能完成(如果并未提供,则使用编译器提供的默认成员赋值行为)。为了避免两次构造,推荐使用类构造函数初始化列表。但很多场合必须使用带有初始化列表的构造函数。例如,成员类型是没有默认构造函数的类,若没有提供显式初始化时,则编译器隐式使用成员类型的默认构造函数,若类没有默认构造函数,则编译器尝试使用默认构造函数将会失败。再例如 `const` 成员或引用类型的成员,因为 `const` 对象或引用类型只能初始化,不能对它们赋值。

7.13.8 类的成员变量的初始化顺序是按照声明顺序吗

在 C++ 中,类的成员变量的初始化顺序只与变量在类中的声明顺序有关,与在构造函数中的初始化列表的顺序无关。而且静态成员变量先于实例变量,父类成员变量先于子类成员变量,父类构造函数先于子类构造函数。

程序代码示例如下:

```
class Test
{
private:
    int n1;
    int n2;
public:
    Test();
};
Test::Test(): n2(2), n1(1)
{ }
```

当查看相关汇编代码时,就能看到正确的初始化顺序了。因为成员变量的初始化次序根变量在内存中的次序有关,而内存中的排列顺序早在编译期就根据变量的定义次序决定了。

从全局看,变量的初始化顺序如下:

- (1) 基类的静态变量或全局变量。
- (2) 派生类的静态变量或全局变量。
- (3) 基类的成员变量。
- (4) 派生类的成员变量。

7.13.9 当一个类为另一个类的成员变量时,如何对其进行初始化

对于类对象数据成员应使用成员初始化列表进行初始化。

程序代码示例如下:

```
class ABC
{
public:
    ABC(int x, int y, int z);
private:
    int a;
    int b;
    int c;
};

class MyClass
{
```

```

public:
    MyClass():abc(1,2,3){}
private:
    ABC abc;
};

```

上例中，因为 ABC 有了显式的带参数的构造函数，那么它是无法依靠编译器生成无参构造函数的，所以没有 3 个 int 型数据，就无法创建 ABC 的对象。

ABC 类对象是 MyClass 的成员，如果要初始化对象 abc，只能用成员初始化列表，没有其他办法将参数传递给 ABC 类构造函数。

7.13.10 C++能设计实现一个不能被继承的类吗

C++不同于 Java，Java 中被 final 关键字修饰的类不能被继承。C++能实现不被继承的类，但是需要自己实现。

为了使类不被继承，最好的办法是使子类不能构造父类的部分，此时子类就无法实例化整个子类。在 C++中，子类的构造函数会自动调用父类的构造函数，子类的析构函数也会自动调用父类的析构函数，所以只要把类的构造函数和析构函数都定义为 private() 函数，那么当一个类试图从它那儿继承时，必然会由于试图调用构造函数、析构函数而导致编译错误。此时该类即不能被继承。

但由此会造成一个问题，private 的构造函数与析构函数无法得到该类的实例。此时可以通过定义静态来创建和释放类的实例。

程序示例如下：

```

class FinalClass1
{
public:
    static FinalClass1* GetInstance()
    {
        return new FinalClass1;
    }
    static void DeleteInstance( FinalClass1* pInstance)
    {
        delete pInstance;
        pInstance = 0;
    }
private:
    FinalClass1() {}
    ~FinalClass1() {}
};

```

在上例中，FinalClass1 这个类是不能被继承的，但是通过该方法得到的实例都位于堆上，需要程序员手动释放。

考虑到这一局限，设计如下一个类。

```

template <typename T> class MakeFinal
{
    friend T;
private:
    MakeFinal() {}
    ~MakeFinal() {}
};

class FinalClass2 : virtual public MakeFinal<FinalClass2>
{
public:

```

```
FinalClass2() {}
~FinalClass2() {}
};
```

上例的 FinalClass2 类使用起来与一般的类没有任何区别，既可以在栈上，也可以在堆上创建实例。而 MakeFinal <FinalClass2> 的构造函数和析构函数都是私有的，但由于类 FinalClass2 是它的友元函数，因此在 FinalClass2 中调用 MakeFinal <FinalClass2> 的构造函数和析构函数也不会造成编译错误。

对于 FinalClass2 类而言，继承一个类并创建它的实例时，会出现编译错误。程序代码示例如下：

```
class Try : public FinalClass2
{
    public :
        Try() {}
        ~Try() {}
};
Try temp;
```

由于类 FinalClass2 是从类 MakeFinal <FinalClass2> 虚继承过来的，在调用 Try 的构造函数时，会直接跳过 FinalClass2，而直接调用 MakeFinal <FinalClass2> 的构造函数。但由于类 Try 不是 MakeFinal <FinalClass2> 的友元，因此不能调用其私有的构造函数。所以，试图从 FinalClass2 继承的类，一旦实例化，都会导致编译错误，因此 FinalClass2 不能被继承。

7.13.11 构造函数没有返回值，那么如何得知对象是否构造成功

这里的“构造”不是单指分配对象本身的内存，而是指在建立对象时做的初始化操作（如打开文件、连接数据库等）。

因为构造函数没有返回值，所以通知对象的构造失败的唯一方法就是在构造函数中抛出异常。构造函数中抛出异常将导致对象的析构函数不被执行，当对象发生部分构造时，已经构造完毕的子对象将会逆序地被析构。

7.13.12 C++中的空类默认产生哪些成员函数

C++中空类默认会产生以下 6 个函数：默认构造函数、复制构造函数、析构函数、赋值运算符重载函数、取址运算符重载函数、const 取址运算符重载函数等。

```
class Empty
{
    public:
        Empty(); // 默认构造函数
        Empty( const Empty& ); // 复制构造函数
        ~Empty(); // 析构函数
        Empty& operator=( const Empty& ); // 赋值运算符
        Empty* operator&(); // 取址运算符
        const Empty* operator&() const; // 取址运算符 const
};
```

默认构造函数和析构函数实际上什么也不做，它们只是用于创建和销毁类的对象。复制构造函数是一种特殊的构造函数，复制构造函数的第一个参数必须为 type X&或 type const X&。要么不存在其他参数，如果存在其他参数，其他参数必须有默认值。

7.13.13 如何设置类的构造函数的可见性

如果不想让外界用户直接构造一个类（假设这个类的名字为 A）的对象，而希望用户只能

构造这个类 A 的子类，那就可以将类 A 的构造函数/析构函数声明为 `protected`，而将类 A 的子类的构造函数/析构函数声明为 `public`。

如果将构造函数/析构函数声明为 `private`，那只有这个类的“内部”的函数才能构造这个类的对象了。这里所说的“内部”是指类的成员函数。

因为在外部不能定义对象，所以不能通过对象调用成员函数，如果想要调用成员函数，可以将成员函数定义为静态，然后通过类的`::`操作符调用，例如，通过如下方式即可：

```
A& ra = A::Instance( );
ra.Print( );
```

7.13.14 public 继承、protected 继承、private 继承的区别是什么

`public`（共有）继承、`protected`（保护）继承和 `private`（私有）继承是常见的 3 种继承方式。

(1) 公有继承。

当采用公有继承时，基类成员对其对象的可见性与一般类及其对象的可见性相同，共有成员可见，其他成员不可见。

基类成员对派生类的可见性对派生类来说，基类的共有成员和保护成员可见；基类的公有成员和保护成员作为派生类的成员时，它们都维持原有的状态；基类的私有成员不可见；基类的私有成员依然是私有的，派生类不可访问和保护成员。

(2) 保护继承。

保护继承与私有继承方式的情况相同。两者的区别在于对派生类的成员而言，基类成员对其对象的可见性与一般类及其对象的可见性相同，共有成员可见，其他成员不可见。

基类成员对派生类的可见性对派生类来说，基类的共有成员和保护成员是可见的：基类的公有成员和保护成员都作为派生类的保护成员，并且不能被这个派生类的子类所访问；基类的私有成员是不可见的；派生类不可访问基类中的私有成员。

基类对象对派生类对象的可见性对派生类对象来说，基类的所有成员都是不可见的。所以，在保护继承时，基类的成员也只能由直接派生类访问，而无法再往下继承。

(3) 私有继承。

在私有继承中，基类成员对其对象的可见性与一般类及其对象的可见性相同，共有成员可见，其他成员不可见。

基类成员对派生类的可见性对派生类来说，基类的共有成员和保护成员是可见的：基类的共有成员和保护成员都作为派生类的私有成员，并且不能被这个派生类的子类所访问；基类的私有成员是不可见的；派生类不可访问基类中的私有成员。基类成员对派生类对象的可见性对派生类对象来说，基类的所有成员都是不可见的。所以，在私有继承时，基类的成员只能由直接派生类访问，而无法再往下继承。表 7-10 所示为成员访问控制列表。

表 7-10 成员访问控制

| 基 类 性 质 | 继 承 性 质 | 派生类性质 |
|-----------|-----------|-----------|
| public | public | Public |
| protected | public | protected |
| private | public | 不能访问 |
| public | protected | protected |

(续)

| 基 类 性 质 | 继 承 性 质 | 派生类性质 |
|-----------|-----------|-----------|
| protected | protected | protected |
| private | protected | 不能访问 |
| public | private | private |
| protected | private | private |
| private | private | 不能访问 |

7.13.15 C++提供默认参数的函数吗

C++可以给函数定义默认参数值。在函数调用时没有指定与形参相对应的实参时，就自动使用默认参数。

默认参数的语法与使用：

- (1) 在函数声明或定义时，直接对参数赋值，这就是默认参数。
- (2) 在函数调用时，省略部分或全部参数。这时可以用默认参数来代替。

通常调用函数时，要为函数的每个参数给定对应的实参。例如：

```
void delay(int loops); //函数声明
void delay(int loops) //函数定义
{
    if(loops==0){
        return;
    }
    for(int i=0;i<loops;i++)
        ;
}
```

上例中，无论何时调用 `delay()` 函数，都必须给函数传递一个值以确定时间，否则无法正确调用函数。此种方法虽然没有什么问题，但当需要用相同的实参反复调用 `delay()` 函数时，反复地传递参数，一方面会不方便，另一方面会造成代码冗余。是否有一种方法可以解决这两种问题了？答案是肯定的，在 C++ 中，可以给函数参数定义默认值，若不给出参数，则按指定的默认值进行工作。例如，在上例中，如果将 `delay()` 函数中的 `loops` 定义成默认值 1000，只需简单地把函数声明改为：`void delay(int loops=1000)`。

这样，以后无论何时调用 `delay()` 函数，都不用给 `loops` 赋值，程序都会自动将它当做值 1000 进行处理。例如，当执行 `delay(2500)` 调用时，`loops` 的参数值为显性化的，被设置为 2500；当执行 `delay()` 时，`loops` 将采用默认值 1000。允许函数默认参数值，是为了让编程简单，让编译器做更多的检查错误工作。

默认参数在函数声明中提供，当有声明又有定义时，定义中不允许默认参数。如果函数只有定义，则默认参数才可出现在函数定义中。例如：

```
void point(int=3,int=4); //声明中给出默认值
void point(int x,int y) //定义中不允许再给出默认值
{
    cout<<x<<endl;
    cout<<y<<endl;
}
```

在使用默认参数时，一般需要注意以下几个问题：

- (1) 如果一个函数中有多个默认参数，则形参分布中，默认参数应从右至左逐渐定义。例如：

1) `void fun(int a=1,int b,int c=3,int d=4);`

2) `void fun(int a,int b=2,int c=3,int d=4);`

上例中,第1种声明方法就是错误的,而第2种声明方法就是正确的。

(2) 在默认参数调用时,调用顺序为从左到右逐个调用。例如,首先声明一个带默认参数的函数 `void mal(int a, int b=3, int c=5)`, 函数调用 `mal(3, 8, 9)` 在调用时有指定参数,则不使用默认参数,是合法调用;函数调用 `mal(3, 5)` 在调用时只指定两个参数,按从左到右的顺序调用,相当于 `mal(3,5,5)`,是合法调用;函数 `mal(3)` 在调用时只指定1个参数,按从左到右的顺序调用,相当于 `mal(5,3,5)`,是合法调用;而函数调用 `mal()` 因为 `a` 没有默认值,所以调用错误;函数调用 `mal(3,, 9)` 应按从左到右的顺序逐个调用,所以也错误。

(3) 默认值可以是全局变量、全局常量,甚至是一个函数。例如:

```
int a=1;
int fun(int);
int g(int x,fun(a));//正确: 允许默认值为函数
```

默认值不可以是局部变量,因此默认参数的函数调用是在编译时确定的,而局部变量的位置与值在编译时均无法确定。例如:

```
void fun( )
{
    int i;
    void g(int x=i);//错误: 处理 g( )函数声明时, i 不可见
}
```

(4) 默认参数可将一系列简单的重载函数合成为一个。例如下面的3个重载函数:

```
void point(int,int){//...}
void point(int a){return point(a,4);}
void point( ){return point(3,4);}
```

可以用下面的默认参数的函数来替代:

```
void point(int=3,int=4);
```

上例中,当调用“`point();`”时,即调用“`point(3,4);`”它是第3个声明的重载函数。当调用“`point(6);`”时,即调用“`point(6,4);`”它是第2个声明的重载函数。当调用“`point(7,8);`”时,即调用第1个声明的重载函数。

如果一组重载函数(可能带有默认参数)都允许相同实参个数的调用,将会引起调用的二义性。例如:

```
void func(int);//重载函数之一
void func(int,int=4);//重载函数之二, 带有默认参数
void func(int=3,int=4);//重载函数三, 带有默认参数
func(7);//错误: 到底调用3个重载函数中的哪个?
func(20,30);//错误: 到底调用后面两个重载函数的哪个?
```

7.13.16 C++中有哪些情况只能用初始化列表而不能用赋值

构造函数初始化列表以一个冒号开始,接着是以逗号分隔的数据成员列表,每个数据成员后面都跟一个放在括号中的初始化式。例如, `Example::Example:ival(0),dval(0.0){}`, 其中 `ival` 与 `dval` 是类的两个数据成员。

在C++中,赋值与初始化列表的原理不一样,赋值是删除原值,赋予新值,初始化列表开辟空间和初始化是同时完成的,直接给予一个值。

所以,在C++赋值与初始化列表的使用情况也不一样,只能用初始化列表而不能用赋值的情况一般有以下3种:

(1) 当类中含有 `const` (常量)、`reference` (引用) 成员变量时,只能初始化不能对它们进

行赋值。常量不能被赋值，只能被初始化，所以必须在初始化列表中完成，C++的引用也一定要初始化，所以必须在初始化列表中完成。

(2) 基类的构造函数都需要初始化列表。构造函数的意思是先开辟空间然后为其赋值，只能算是赋值，不算初始化。

(3) 成员类型是没有默认构造函数的类。若没有提供显式初始化式，则编译器隐式使用成员类型的默认构造函数，若类没有默认构造函数，则编译器尝试使用默认构造函数将会失败。

7.14 虚函数

虚函数中的“虚”并不是实际生活中虚拟的意思，因为没有“实”函数的说法。虚函数是面向对象编程中函数的一种特定形态，是C++中用于实现多态的一种有效机制。

7.14.1 什么是虚函数

指向基类的指针在操作它的多态类对象时，会根据不同的类对象调用其相应的函数，这个函数就是虚函数，虚函数用 `virtual` 修饰函数名。虚函数的作用是在程序的运行阶段动态地选择合适的成员函数，在定义了虚函数后，可以在基类的派生类中对虚函数进行重新定义。在派生类中重新定义的函数应与虚函数具有相同的形参个数和形参类型（参数类型的顺序也要一致），以实现统一的接口。如果在派生类中没有对虚函数重新定义，则它继承其基类的虚函数。

在基类的类定义中定义虚函数的一般形式如下：

```
class <类名>
{
    virtual 函数返回值类型 虚函数名(形参表);
    ...
};
```

下述程序示例代码是一个虚函数的例子。

```
#include<iostream>
using namespace std;

class A
{
public:
    virtual void Print()
    {
        printf("This is Class A\n");
    }
};

class B : public A
{
public:
    void Print()
    {
        printf("This is Class B\n!");
    }
};

int main()
{
```

```

    A a;
    B b;
    A* p1=&a;
    A* p2=&b;
    p1->Print();
    p2->Print();
    return 0;
}

```

程序输出结果:

```

This is Class A
This is Class B

```

需要注意的是，虚函数虽然非常好用，但是在使用虚函数时，并非所有的函数都需要定义成虚函数，因为实现虚函数是有代价的。在使用虚函数时，需要注意以下几个方面的内容：

(1) 只需要在声明函数的类体中使用关键字 `virtual` 将函数声明为虚函数，而定义函数时不需要使用关键字 `virtual`。

(2) 当将基类中的某一成员函数声明为虚函数后，派生类中的同名函数自动成为虚函数。

(3) 如果声明了某个成员函数为虚函数，则在该类中不能出现与这个成员函数同名并且返回值、参数个数、类型都相同的非虚函数。在以该类为基类的派生类中，也不能出现这种同名函数。

(4) 非类的成员函数不能定义为虚函数，全局函数以及类的成员函数中静态成员函数和构造函数也不能定义为虚函数，但可以将析构函数定义为虚函数。将基类的析构函数定义为虚函数后，当利用 `delete` 删除一个指向派生类定义的对象指针时，系统会调用相应的类的析构函数。而不将析构函数定义为虚函数时，只调用基类的析构函数。

(5) 普通派生类对象，先调用基类构造再调用派生类构造。

(6) 基类的析构函数应该定义为虚函数，这样可以在实现多态的时候不造成内存泄漏。基类析构函数未声明 `virtual`，基类指针指向派生类时，`delete` 指针不调用派生类析构函数。有 `virtual`，则先调用派生类析构再调用基类析构。

(7) 基类指针动态建立派生类对象，普通调用派生类构造函数。

(8) 指针声明不调用构造函数。

虚函数的使用可以极大地提高软件开发的效率，那么虚函数是通过什么实现的呢？其实，虚函数是通过一张虚函数表（Virtual Table）来实现的。该表是一个类的虚函数的地址表，解决了继承、覆盖的问题，保证它能真实反应实际的函数。这样，在有虚函数的类的实例中，这个表被分配在了这个实例的内存中，所以当用父类的指针来操作一个子类的时候，这张虚函数表就显得非常重要，它指明了实际所应该调用的函数。

C++的编译器能够保证虚函数表的指针存在于对象实例中最前面的位置，通过对象实例的地址得到这张虚函数表，然后就可以遍历其中的函数指针，并调用相应的函数。例如，有这样的一个类：

```

class Base
{
public:
    virtual void f() { cout << "Base::f" << endl; }
    virtual void g() { cout << "Base::g" << endl; }
    virtual void h() { cout << "Base::h" << endl; }
};

```


可以通过 Base 的实例来得到虚函数表。程序示例如下：

```
typedef void(*Fun)(void);
Base b;
Fun pFun = NULL;
cout << "虚函数表地址: " << (int*)&b << endl;
cout << "虚函数表 — 第一个函数地址: " << (int*)((int*)&b) << endl;
pFun = (Fun)((int*)((int*)&b));
pFun();
```

实际运行结果如下：

```
虚函数表地址: 0012FED4
虚函数表—第一个函数地址: 0044F148
Base::f
```

通过这个示例可以看到，可以通过强行把&b转成int*，取得虚函数表的地址，然后再次取址就可以得到第一个虚函数的地址了，也就是Base::f()，这在上面的程序中得到了验证（把int*强制转成了函数指针）。通过这个示例可以知道，调用Base::g()和Base::h()的代码如下：

```
(Fun)((int*)((int*)&b)+0); // Base::f()
(Fun)((int*)((int*)&b)+1); // Base::g()
(Fun)((int*)((int*)&b)+2); // Base::h()
```

应在构造函数中进行虚函数表的创建和虚函数指针的初始化。根据构造函数的调用顺序，在构造子类对象时，要先调用父类的构造函数，此时编译器只“看到了”父类，并不知道后面是否还有继承者，它初始化父类对象的虚函数表的指针，该虚函数表指针指向父类的虚函数表。当执行子类的构造函数时，子类对象的虚函数表指针被初始化，指向自身的虚函数表。

编译器发现一个类中有虚函数，便会立即为此类生成虚函数表，虚函数表的各表项为指向对应虚函数的指针。编译器还会在此类中隐含插入一个指针vptr（对VC编译器来说，它插在类的第一个位置上）指向虚函数表。调用此类的构造函数时，在类的构造函数中，编译器会隐含执行vptr与vtable的关联代码，将vptr指向对应的vtable，将类与此类的vtable联系起来，另外在调用类的构造函数时，指向基础类的指针此时已经变成指向具体的类的this指针，这样依靠此this指针即可得到正确的vtable。这样才能真正与函数体进行连接，这就是动态联编，实现多态的基本原理。

7.14.2 C++如何实现多态

C++中通过虚函数实现多态。虚函数的本质就是通过基类访问派生类定义的函数。每一个含有虚函数的类，其实例对象内部都有一个虚函数表指针。该虚函数表指针被初始化为本类的虚函数表的内存地址。所以在程序中，不管对象类型如何转换，但该对象内部的虚函数表指针是固定的，这样才能实现动态地对对象函数进行调用，这就是C++多态性的原理。

```
#include <iostream>
using namespace std;

class A
{
public:
    A() {}
    virtual void foo()
    {
        cout << "This is A" << endl;
    }
};
```

```

class B : public A
{
    public:
        B() {}
        void foo()
        {
            cout << "This is B" << endl;
        }
};

int main()
{
    A *a = new B();
    a->foo();
    return 0;
}

```

程序输出结果:

This is B

如果将类 A 中 foo() 函数前的 virtual 关键字去掉, 则程序输出结果如下:

This is A

7.14.3 C++中继承、虚函数、纯虚函数分别指的是什么

继承是指一种事物自动获得另一种事物的全部东西(属性, 能力)。在 C++ 中继承的使用方式如下: class 派生类名: <public|protected|private>基类名 {}, 3 种继承方式跟类成员的 3 种访问属性一样。

用 virtual 修饰的函数就是虚函数。如果需要使用多态特性, 就必须使用虚函数。以基类对象的身份调用的虚函数, 如果对象是派生类的, 派生类的对应函数会被调用, 从而可以实现通过完全相同的调用形式让不同的类型的对象作为自己不同的响应。

纯虚函数是一种特殊的虚函数, 格式一般如下:

```

class <类名>
{
    virtual( )函数返回值类型 虚函数名(形参表)=0;
    ...
};

class <类名>

```

由于在很多情况下, 基类中不能对虚函数给出有意义的实现, 只能把函数的实现留给该基类的派生类去做。例如, 动物作为一个基类可以派生出老虎、孔雀等子类, 但是动物本身生成对象不合情理。此时就可以将函数定义为纯虚函数(方法: virtual ReturnType Function()), 编译器要求存在若干派生的非抽象类, 则在派生类中必须予以重载以实现多态性。

对于纯虚函数, 编译器要求在派生类中予以重载以实现多态性。含有纯虚函数的类称为抽象类, 抽象类不能生成对象。纯虚函数永远不会被调用, 它们主要用来统一管理子类对象。

7.14.4 C++中的多态种类有哪几种

C++ 中的多态种类包括参数多态、引用多态、重载多态以及强制多态等。

参数多态是指采用参数化模板, 通过给定不同的类型参数, 使得一个结构有多种类型、模板。引用多态是指同样的操作可以用于一个类型及其子类型。重载多态是指同一个名字在不同的上下文中有不同的类型。而强制多态则是指把操作对象的类型强加以变换, 以符合函数或操作符的要求。

7.14.5 什么函数不能声明为虚函数

一个类中将所有的成员函数都尽可能地设置为虚函数总是有益的，但是设置虚函数需要注意以下5个方面的内容：

(1) 只有类的成员函数才能说明为虚函数。

(2) 静态成员函数不能为虚函数，因为调用静态成员函数不要实例，但调用虚函数需要一个实例，两者相互矛盾。

(3) 内联函数不能为虚函数。

(4) 构造函数不能为虚函数。

(5) 析构函数可以为虚函数，而且通常声明为虚函数。

构造函数不能是虚函数，是因为构造函数是在对象完全构造之前运行的，换句话说，运行构造函数前，对象还没有生成，更谈不上动态类型了。构造函数是初始化虚表指针，而虚函数放到虚表里面，当要调用虚函数的时候首先要知道虚表指针，这个就存在矛盾的地方了，所以构造函数不可能是虚函数。构造函数虽然不能是虚函数，但构造函数里是可以调用虚函数的。程序示例如下：

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base()
    {
        f();
    }
    virtual void f()
    {
        cout<<"base"<<endl;
    }
};

class Derived : public Base
{
public:
    Derived() {}
    void f()
    {
        cout<<"Derived"<<endl;
    }
};

int main()
{
    Base * p=new Derived;
    p->f();
    return 0;
}
```

程序输出结果：

```
base
Derived
```

base
Derived

上例中, `Base * p=new Derived`, 基类的指针生成了一个派生类对象, 将会隐式调用 `base` 的构造函数, 尽管对象是 `Derived`, 但构造基类部分时, 还只是个 `Base`, 所以会调用基类的虚函数 `f()`。

析构函数可以是虚函数, 而且有的时候是必须的, 基类指针指向派生类, 用基类指针 `delete` 时, 如果不定义成虚函数, 派生类中派生的那部分无法析构。

析构函数执行时先调用派生类的析构函数, 然后才调用基类的析构函数。如果析构函数不是虚函数, 而程序执行时又要通过基类的指针去销毁派生类的动态对象, 那么用 `delete` 销毁对象时, 只调用了基类的析构函数, 未调用派生类的析构函数。这样会造成销毁对象不完全。程序示例如下:

```
#include<iostream>
using namespace std;

class CPerson
{
public:
    virtual ~CPerson( );
protected:
    char * m_lpszName;
    char * m_lpszSex;
};

class CStudent:public CPerson
{
public:
    ~CStudent( );
protected:
    int m_iNumjber;
};

CPerson::~~CPerson( )
{
    cout<<"~CPerson!"<<endl;
}

CStudent::~~CStudent( )
{
    cout<<"~CStudent!"<<endl;
}

int main( )
{
    CPerson * poCPerson = new CStudent;
    if(NULL==poCPerson)
    {
        exit(0);
    }
    delete poCPerson;
    cout<<"CStudent 对象已经完成析构"<<endl;
    CStudent oCStudent;
    return 0;
}
```

程序输出结果:

```

~CStudent!
~CPerson!
CStudent 对象已经完成析构
~CStudent!
~CPerson!

```

7.14.6 是否可以把每个函数都声明为虚函数

虽然虚函数很有效，但是不可以把每个函数都声明为虚函数。因为使用虚函数是要付出代价的。由于每个虚函数的对象在内存中都必须维护一个虚函数表，因此在使用虚函数时，尽管带来了使用的方便，却会额外产生一个系统开销。如果仅是一个很小的类，且不想派生其他类，那么根本没有必要使用虚函数。

7.14.7 C++中如何阻止一个类被实例化

C++中可以通过使用抽象类，或者将构造函数声明为 `private` 阻止一个类被实例化。抽象类之所以不能被实例化，是因为抽象类不能代表一类具体的事物，它是对多种具有相似性的具体事物的共同特征的一种抽象。例如，声明一个抽象类车，但是却不能用这个类来创造某个具体的事物来，只能派生一个汽车，才可以产生出来。

引申：

(1) 一般在什么时候将构造函数声明为 `private`？

例如，要阻止编译器生成默认的复制构造函数的时候。

(2) 什么时候编译器会生成默认的复制构造函数？

只要自己没写，而程序需要，都会生成。

(3) 如果已经写了一个构造函数，编译器还会生成复制构造函数吗？
会。

7.15 编程技巧

编程，容易；技巧，容易；编程技巧，不容易。

7.15.1 当 `while()` 的循环条件是赋值语句时会出现什么情况

在 `while()` 的循环条件里面定义一个变量并赋值为 0，程序代码如下：

```

while(int i=0)
{
    printf("%d\n",i);
    i--;
}

```

以上代码不执行任何动作，相当于执行了 `while(0)` 操作，循环结束，`while` 循环体不执行。而在 `while` 的循环条件里面定义一个变量并赋值为非 0 时，相当于执行了 `while(1)`，程序进入无限循环。

```

while(int i=1)
{
    printf("%d\n",i);
    i--;
}

```

需要注意的是，上述代码之所以不停地输出为 1，而不是执行 `i--`，是因为在 `while` 循环

条件里，重新定义了一个局部变量 *i*，对其进行了重新赋值，所以 *i* 的初始值一直为 1，而不是自减。

7.15.2 不使用 if/?:/switch 及其他判断语句如何找出两个 int 型变量中的最大值和最小值

寻找两个变量中的较大值，一般可以用判断型语句进行比较，如 `if(a>b)`，按照题目中的要求，不能采用判断语句，所以可以利用绝对值的方法或是移位的方法。

方法一：

```
int max=((a+b)+abs(a-b))/2
```

如果 $a > b$ ，则 $\text{max} = a$ ；如果 $a < b$ ，则 $\text{max} = b$ 。

```
int min=((a+b)-abs(a-b))/2
```

如果 $a > b$ ，则 $\text{min} = b$ ；如果 $a < b$ ，则 $\text{min} = a$ 。

上例中，`abs` 是 C 语言中的用于求绝对值的函数，C 语言中还有一个类似的函数，函数名为 `fabs`。

方法二：对变量的差值进行移位操作，通过其是否为非 0 值确定两个变量的大小。

```
#include <stdio.h>
int main( )
{
    int a=1,b=4;
    int c=a-b;
    int MAX=(unsigned)c>>(sizeof(int)*8-1);
    if(!MAX)
        printf("%d\n",a);
    else
        printf("%d\n",b);
    return(0);
}
```

程序的输出结果：

4

方法三：通过移位操作来判断。程序示例如下：

```
void compare(int a, int b)
{
    static char* op[] = { "=", "<", ">" };
    int i = (unsigned(a-b) >> 31) + (unsigned(b-a) >> 31)*2;
    printf( "%d %s %d\n ", a, op[i], b);
}
```

上例中，满足：

(1) 如果 $a > b$ ，

那么 $(\text{unsigned}(a-b) >> 31) = 0$ ， $(\text{unsigned}(b-a) >> 31)*2 = 2$ 。

(2) 如果 $a = b$ ，

那么 $(\text{unsigned}(a-b) >> 31) = 0$ ， $(\text{unsigned}(b-a) >> 31)*2 = 0$ 。

(3) 如果 $a < b$ ，

那么 $(\text{unsigned}(a-b) >> 31) = 1$ ， $(\text{unsigned}(b-a) >> 31)*2 = 0$ 。

方法四：通过加减运算与移位运算结合的方式实现。

```
void compare3(int a, int b)
{
```

```

    int min = a+(((b-a)>>31)&(b-a));
    int max = a-(((a-b)>>31)&(a-b));
    cout<<min<<endl;
    cout<<max<<endl;
}

```

注意：正数的补码和原码相同。负数的补码是正数的补码按位取反，末位加一。

7.15.3 C 语言获取文件大小的函数是什么

某些标识符是预定义的，扩展后将生成特定的信息，它们同预处理器表达式运算符#define一样，不能取消定义或重新定义。预定义标识符表见表 7-11。

表 7-11 预定义标识符

| 函 数 | 描 述 |
|----------------------------------|---|
| __FILE__ | 包含当前程序文件名的字符串，包含了详细路径，如 G:/program/study/c+/test1.c |
| __LINE__ | 包含当前源文件行数的十进制常量 |
| __DATE__ | 包含编译日期的字符串字面值 |
| __STDC__ | 如果编译器遵循 ANSI C 标准，它就是个非零值 |
| __TIME__ | 包含编译时间的字符串字面值 |
| __FUNC__（在有些编译系统中为 __FUNCTION__） | 当前所在函数名，在编译器的较高版本中支持 |

其中，标识符 __LINE__ 和 __FILE__ 通常用来调试程序，标识符 __DATE__ 和 __TIME__ 通常用来在边以后的程序中加入一个时间标志，以区分程序的不同版本。当要求程序严格遵循 ANSIC 标准时，标识符 __STDC__ 就会被赋值为 1。

7.15.4 表达式 a>b>c 是什么意思

在弄清这个问题前，先看如下代码：

```

#include <stdio.h>

int main()
{
    int a=5,b=4,c=3;
    printf("%d\n",a>b>c);
    return 0;
}

```

程序输出结果：

0

在上例中，a>b>c 到底是如何执行的呢？对于这种连续运算，根据优先级，首先进行 a>b 的比较判断，本例中 a>b 为真，所以返回值为 1，接着比较该返回值与 c 的大小。因为 c 的值为 3，1>c 表达式为假，所以返回值为 0。所以，最终的输出为 0。

对于赋值运算符，结果又如何呢？以如下程序为例。

```

#include <stdio.h>

int main()
{
    int b,c;
    int a=(b=(c=020)&&(1==2));
}

```

```
printf("%d %d %d\n",a,b,c);
return 0;
}
```

程序输出结果:

0 0 16

在赋值语句中, $c=020$, 因为以 0 开头的数字一般表示的都是八进制的数值, 所以折合成十进制的数为 16。根据优先级关系, b 的值为 $(c=020) \&\& (1==2)$ 的结果, 由于 $c=020$ 是一个赋值语句, 所以该赋值语句的返回值为真, 即为 1, 而 $1==2$ 则为假, 返回值为 0, 所以 b 的值为 0, $a=(b=0)$, 所以 a 的值为 0。

7.15.5 如何打印自身代码

用常规的 `printf` 语句输出是得不到与自身代码一模一样的结果的, 因为这涉及一个自身嵌套的问题。如果希望打印程序自身代码, 可以参考如下实现。

```
#include <stdio.h>
int main()
{
    char *p = "#include <stdio.h>%cint main( )%c{%c      char *p = %c%s%c;%c\n";
    printf(p,10,10,10,34,p,34,10,10,10);%c    return 0;%c}";
    printf(p,10,10,10,34,p,34,10,10,10);
    return 0;
}
```

上面代码的实现需要注意以下几个方面的内容:

(1) 写好一个程序。

(2) 定义一个字符串 `str` 把原来的代码抄进去; 不能显示的字符和特殊字符都用 `%c` 替换, 如换行、引号等。

用一个输出语句 `printf` 打印 `str`。注意这里, 格式控制的时候, 10 表示换行, 34 表示", 92 表示\, 110 表示 `n`, 9 表示 `t`。

7.15.6 如何实现一个最简单病毒

可以把最简单的病毒理解为一个无限运行的恶意程序, 无限运行可以通过无限循环实现, 而恶意可以通过申请内存空间来实现, 所以可以用如下代码来实现一个最简单的病毒。

```
while(1)
{
    int *p=new int [10000000];
}
```

该代码首先新建一个无限循环, 然后在循环内执行一个内存申请操作, 最终系统内存会被该程序占用完, 导致系统出现宕机的情况。

引申: 嵌入式系统中经常要用到无限循环, 怎样用 C 语言编写无限循环?

有多种执行无限循环的方式, 第一种是使用 `while`, 将 `while` 条件置为 1, 具体使用方式如下所示。

```
while(1)
{
}
```

除了使用 `while` 以外, 也可以使用 `for` 循环, `for` 语句为空, 因为没有循环区间, 没有循环条件, 也没有条件语句, 所以能够执行无限循环。具体使用方式如下所示。

```
for(;;)
{
}
```

使用 for 循环的上述方式表示无限循环时，也可以将循环条件设置为 1。方式如下所示。

```
for(;;1;)
{
}
```

除了以上两种方法以外，还有一种不推荐使用的方法——使用 goto 语句，程序示例如下所示。

```
Loop:
...
goto Loop;
```

goto 语句一般不推荐使用，但是在必要的嵌入式环境下也不失为一种不错的解决方案。

7.15.7 如何只使用一条语句实现 x 是否为 2 的若干次幂的判断

如果一个数是 2 的若干次幂，那么其二进制表示中最高位为 1，其他位为 0，该数减去 1 之后的数的二进制表示为全 1，所以将两数进行与操作，判断其最终结果是否为 0，可以只用一语句实现判断该数是否是 2 的若干次幂的功能。

程序示例如下：

```
int i = 512;
cout << boolalpha << ((i & (i - 1)) ? false : true) << endl;
```

7.15.8 如何定义一对相互引用的结构

用常规的定义相互引用的结构一般容易出现类似于死锁一样的问题。例如，A 引用 B 的成员，B 也引用 A 的成员，由于 C 语言需要先声明后使用，所以常规的方法会引起编译器错误。

程序示例如下：

```
typedef struct
{
    int afield;
    BPER bpointer;
}*APTR;
typedef struct
{
    int bfield;
    APTR apointer;
}*BPTR;
```

上例中 BPER 没有被定义就已经被使用了，所以错误，需要采取其他的方式来实现。指针就是一种不错的方式，以下方式都可以实现定义一对相互引用的结构。

```
struct b;//此处使用结构体的不完整声明
struct a
{
    int afield;
    struct b* bpointer;//此处结构体虽然未定义，但是编译器却可以接受
};
struct b
{
    int bfield;
    struct a* apointer;
};
```

需要注意一个区别：结构体的自引用（self reference）就是在结构体内部，包含指向自身类型结构体的指针。结构体的相互引用（mutual reference）即在多个结构体中，都包含指向其他结构体的指针。

```
struct a
{
    int b;
    int c;
    char d;
    struct a z;
    struct a *p;
}
```

上述结构就有问题，因为结构中不能定义结构本身的非指针变量，如果编译器支持则会导致无限嵌套，因此一般编译器都会认为 `struct a` 是未定义的类型，即使提前声明也不会有任何用处。

7.15.9 什么是逗号表达式

关于二维数组赋值的一个陷阱：

```
int a[3][2]={{0,1},{2,3},{4,5}};
int *p;
p=a[0];
printf("%d\n",p[0]);
```

程序的输出结果：

1

程序示例如下：

```
int a[3][2]={{0,1},{2,3},{4,5}};
int *p;
p=a[0];
printf("%d\n",p[0]);
```

程序的输出结果：

0

上述两段代码很类似，为什么输出却有这么大的不同？两者的区别在于二维数组的初始化问题，第一种情况由于是逗号表达式，所以数组的元素等价于 $\{\{1,3\},\{5,0\},\{0,0\}\}$ ，与第二种情况不一样。

C 语言提供一种特殊的运算符——逗号运算符，优先级最低，它将两式连接起来。例如， $(3+4, 2+8)$ 称为逗号表达式，其求解过程是先求表达式 1 的运算结果，然后求解表达式 2 的结果，而整个表达式的值为表达式 2 的值，如 $(3+4, 2+8)$ 的值是 10。 $(a=3*5, a*4)$ 的值为 60。

在使用逗号表达式时，首先需要弄清楚其表示形式，其表示形式如下：

表达式 1，表达式 2，表达式 3……表达式 n

使用逗号表达式，需要注意以下 3 个方面的事项：

- (1) 逗号表达式的运算过程为从左往右逐个计算表达式。
- (2) 逗号表达式作为一个整体，它的值为最后一个表达式（也即表达式 n）的值。
- (3) 逗号运算符的优先级在所有运算符中最低。

例如，语句 `i = 1, 2;` 虽然是逗号表达式，但是赋值符的优先级更高，所以 `i` 的值为 1，接着执行常量 2 的运算，运算结果会被丢弃。在编译器中，虽然这种语句的写法是合法的，但是是不提倡。再例如，语句 `(a=3*5, a*4)` 和语句 `b=(a=3*5, a*4)` 意义不一样，第一个语句中 `a` 的值为 15，第二个语句 `a` 的值为 15，但 `b` 的值为 60。

引申: `int i=(j=4,k=8,l=16,m=32)`, 则 `i` 的值是多少?

输出结果为 32。当一个语句是由多个被逗号运算符隔开的表达式组成时, 此语句的值为最后一个表达式的值。

首先看一个例子, `int i=(j=4)`, 它等价于 `i=j=4` 语句, 即 `int j = 4` 与 `int i = j`。而 `int i=(j=4,k=8,l=16,m=32)`则等价于: `int j=4, k=8, l=16, m=32, i=m=32`, 所以输出为 32。例如, `int a=3`, 则执行 `a+=a-=a+=a*a` 运算后, `a` 的值变为 0, 因为这个连续运算语句可以转换为以下 3 个语句: 首先计算 `a+=a*a`, 把 `a` 的值 3 带入, `a+=3+3*3`, 则 `a` 的值变为 12, 然后计算 `a-=a`, 此时 `a` 的值变为 0, 最后执行 `a+=a` 操作, `a` 的值最终变为 0。

7.15.10 `\n` 是否与 `\n\r` 等价

换行(`\n`)就是光标下移一行却不会移到这一行的开头, 回车(`\r`)就是回到当前行的开头却不向下移一行。

按〈Enter〉键后会执行"`\n\r`", 这样就是看到的一般意义的回车了, 所以在用 16 进制文件查看方式看一个文本, 就会在行尾发现"`\n\r`"。

Tab 是制表符, 就是"`\t`", 作用是预留 8 个字符的显示宽度, 用于对齐。

引申: "`\n`"与"`\n'`"是否有区别?

两者存在区别, "`\n`"是一个字符串, 该字符串以'`'0'`结束, 即它实际包含了两个字符, 而"`\n'`"只是一个简单的字符而已, 所以两者不相等。

7.15.11 什么是短路求值

短路求值是常见的计算机问题, 所谓短路求值即对于 (条件 1 && 条件 2), 如果“条件 1”是 `false`, 那“条件 2”的表达式会被忽略。对于 (条件 1 || 条件 2), 如果“条件 1”为 `true`, 而“条件 2”的表达式则被忽略了。

程序示例如下:

```
#include <stdio.h>
int main()
{
    int i = 6, j = 1;
    if(i > 0 || (j++) > 0)
        ;
    printf("%d\n", j);
    return 0;
}
```

程序输出结果:

1

输出为什么不是 2 而是 1 呢? 其实, 这里就涉及一个短路计算的问题。由于 `if` 语句是一个条件判断语句, 里面是有两个简单语句进行或运算组合的复合语句, 因为或运算中, 只要参与或运算的两个表达式的值都为真, 则整个运算结果为真, 而由于变量 `i` 的值为 6, 已经大于 0 了, 而该语句已经为 `true`, 则不需要执行后续的 `j++` 操作来判断真假, 所以后续的 `j++` 操作不执行, `j` 的值仍然为 1。

程序示例如下:

```
#include <stdio.h>
int main()
{
    int a=5, b=6, c=7, d=8, m=2, n=2;
```

```

(m=a>b)&&(n=c>d);
printf("%d\n",n);
return 0;
}

```

程序的输出结果:

2

因为短路计算的问题,对于&&操作,由于两个表达式的值如果有一个为假,则整个表达式的值都为假,如果前一个语句的返回值为 false,则无论后一个语句是真是假,整个条件判断都为假。不用执行后一个语句,而 a>b 为 false,程序不执行 n=c>d,所以 n 的值保持为初值 2。

7.15.12 已知随机数函数 rand7(), 如何构造 rand10()函数

要保证 rand10()产生的随机数是整数 1~10 的均匀分布,可以构造一个 1~10*n 的均匀分布的随机整数区间 (n 为任意正整数)。假设 x 是这个 1~10*n 区间上的一个随机数,那么 x%10+1 就是均匀分布在 1~10 区间上的整数。

根据题意,rand7()函数返回 1~7 的随机数,那么 rand7()-1 则得到一个离散整数集合,该集合为 {0, 1, 2, 3, 4, 5, 6}, 该集合中每个整数的出现概率都为 1/7。那么(rand7()-1)*7 得到另一个离散整数集合 A, 该集合元素为 7 的整数倍,即 {0, 7, 14, 21, 28, 35, 42}, 其中每个整数的出现概率也都为 1/7。而由于 rand7()得到的集合 B={1, 2, 3, 4, 5, 6, 7}, 其中每个整数出现的概率也为 1/7。显然集合 A 与集合 B 中任何两个元素组合可以与 1~49 之间的一个整数一一对应,即 1~49 之间的任何一个数,可以唯一确定 A 和 B 中两个元素的一种组合方式,反过来也成立。由于 A 和 B 中元素可以看成是独立事件,根据独立事件的概率公式 $P(AB)=P(A)P(B)$, 得到每个组合的概率是 $1/7*1/7=1/49$ 。因此, (rand7()-1)*7+rand7()生成的整数均匀分布在 1~49 之间,每个数的概率都是 1/49。

所以(rand7()-1)*7+rand7()可以构造出均匀分布在 1~49 的随机数,为了将 49 种组合映射为 1~10 之间的 10 种随机数,就需要进行截断了,即将 41~49 这样的随机数剔除掉,得到的数 1~40 仍然是均匀分布在 1~40 的,这是因为每个数都可以看成一个独立事件。

程序代码如下:

```

#include <iostream>
#include <ctime>
using namespace std;

int rand7()
{
    return rand()%7+1;
}

int rand10()
{
    int x = 0;
    do
    {
        x = (rand7()-1)*7 + rand7();
    } while (x>40);
    return x%10+1;
}

int main()

```

```

{
    srand(unsigned(time(0)));
    for(int i = 0; i != 10; ++i)
        cout<<rand10()<<" ";
    cout<<endl;
    return 0;
}

```

程序输出结果

9 7 10 8 7 8 9 4 5 8

7.15.13 printf("%p\n",(void *)x)与 printf("%p\n",&x)有何区别

printf("%p\n",(void *)x);语句打印 x 被转为指针的地址,就是它的值。printf("%p\n",&x);将打印变量 x 的地址。

当整型变量 x 的值为 0x87654321 时,两者的输出分别为

87654321
0012FF60

7.15.14 printf()函数是否有返回值

有。printf()函数的一般形式为 int printf(const char* format, [argument]), 它返回一个 int 值,表示被打印的字符数。

程序示例如下:

```

#include <iostream>
using namespace std;

int main()
{
    int i=4321;
    printf("%d\n",printf("%d\n",printf("%d\n",i)));
    return 0;
}

```

程序输出结果如下:

4321
5
2

程序之所以首先输出为 4321,是因为 printf 打印的目标为整型变量 i 的值,由于 i 的值包含 4 个字符数,而'\n'占据 1 个字符,所以一共占据了 5 个字符,所以第二行打印为 5,5 与'\n'合计为两个字符,所以第三行输出为 2。

7.15.15 不能使用任何变量,如何实现计算字符串长度函数

Strlen()

递归是一种自己调用自己的方式,有点像死循环的嵌套。如果题目没有要求,最容易想到的方法是使用一个 for 循环,对字符串进行遍历,当遇到'\0'结束,最终记录字符串的长度,程序示例如下:

```

#include<stdio.h>

int Strlen(const char *str) /* 使用了一个 int 型变量 len*/
{
    int len = 0;

```

```

        if(str==NULL)
            return 0;
        for(; *str++ != '\0';)
        {
            len++;
        }
        return len;
    }

    int main( )
    {
        printf("%d\n",Strlen("amc"));
        return 0;
    }

```

程序的输出结果:

3

上例中, 使用了临时变量, 因为题目要求不能使用任何变量, 所以上述方法不可取, 可以采用递归的思想来实现。程序示例如下:

```

#include<stdio.h>

int Strlen(const char* s)
{
    if(*s!='\0')
        return 1+Strlen1(++s);
    else
        return 0;
}

int main( )
{
    printf("%d\n",Strlen("amc"));
    return 0;
}

```

程序的输出结果

3

还可以将上述递归方式简化为如下所示的表现形式:

```

int Strlen(const char* s)
{
    return *s=='\0'?0:(1+Strlen2(++s));
}

```

7.15.16 负数除法与正数除法的运算原理是否一样

在 C 语言中, 负数除法运算与正数除法运算不一样, 主要遵循以下规则: 除号的正负取舍和一般的算数一样, 符号相同为正, 相异为负, 求余符号的正负取舍和被除数符号相同。

以-3/16, 16/-3; -3%16, 16%-3 为例分析, $-3/16 = 0$, $16/-3 = -5$ 。 $-3\%16 = -3$, $16\%-3 = 1$ 。

7.15.17 main()主函数执行完毕后, 是否可能会再执行一段代码

可以。例如, 可以用_onexit 注册一个函数, 它会在 main 之后执行 int fn(void)。需要注意的是, 使用_onexit()函数需要添加头文件 stdlib.h, 否则会报编译错误。

程序示例如下:

```
#include <stdio.h>
#include <stdlib.h>

int fn( )
{
    printf( "next\n" );
    return 0;
}

int main( )
{
    _onexit(fn);
    printf( "This is executed first\n" );
    return 0;
}
```

程序输出结果:

```
This is executed first
next
```

需要注意的是, 在 C 语言中, 用户代码也可以调用 `main()` 函数, 程序示例如下:

```
#include <stdio.h>
#include <stdlib.h>
void main( );

void test( )
{
    main( );
}

void main( )
{
    printf("test");
    test( );
}
```

上例中, 程序运行一段时间会中断, 递归却没有终止。

数据库

第 8 章

在同一个数据集合中，不同的选择条件对应了不同的输出结果，数据库就是这样一种按数据结构来组织、存储和管理数据的仓库。程序或用户可以通过它来进行数据的访问与修改，它是数据存储的灵魂，很多大型的数据库企业都比较关注求职者对数据库的掌握程度，通过考查基础知识、SQL 语句等判断其是否具备企业的要求。

8.1 数据库概念

数据管理经历了人工管理、文件系统到数据库系统 3 个阶段。数据库是具有逻辑关系和确定意义的数据结合，它能克服传统文件组织中所产生的一系列问题，数据冗余小，由于关系型数据库管理系统对于信息查询具有很大的灵活性，并且设计简单，所以已经被广泛使用在了实际的系统开发中。

8.1.1 关系数据库系统与文件数据库系统有什么区别

关系数据库（relational database）是一个被组织成一组正式描述的表格的数据项的集合，这些表格中的数据能以不同的方式被存取或重新召集而不需要重新组织数据库表格，它对应于一个关系模型中的所有关系的集合。例如，教师、学生、课程这些关系以及关系间的联系就组成了一个教务系统。文件数据库系统是对文件的操作，包括存储、查询等。

关系数据库系统与文件数据库系统的区别如下：

- （1）关系数据库系统的主要特征是数据的结构化，而文件数据库系统是数据的非结构化。
- （2）关系数据库系统中，用户看到的逻辑结构是二维表，而文件数据库系统中，基本元素是文件。
- （3）文件数据库系统可以实现多媒体文件管理，支持 C/S 工作模式。

8.1.2 SQL 语言的功能有哪些

SQL 是结构化查询语言（Structured Query Language）的缩写，其功能包括数据查询、数据操作、数据定义和数据控制 4 个部分。

数据查询是数据库中最常见的操作，通过 select 语句可以得到所需的信息。SQL 语言的数据操作语句（Data Manipulation Language, DML）主要包括插入数据、修改数据以及删除数据 3 种语句。SQL 语言使用数据定义语言（Data Definition Language, DDL）实现数据定义功能，可对数据库用户、基本表、视图、索引进行定义与撤销。数据控制语句（Data Control Language, DCL）用于对数据库进行统一的控制管理，保证数据在多用户共享的情况下能够安全。

基本的 SQL 语句有 select、insert、update、delete、create、drop、grant、revoke 等。其具体使用方式见表 8-1。

表 8-1 基本查询语句

| | 关 键 字 | 描 述 | 语 法 格 式 |
|------|--------|-----------|---|
| 数据查询 | select | 选择符合条件的记录 | select * from table where 条件语句 |
| 数据操作 | insert | 插入一条记录 | insert into table(字段 1, 字段 2...)values(值 1, 值 2...) |
| | update | 更新语句 | update table set 字段名=字段值 where 条件表达式 |
| | delete | 删除记录 | Delete from table where 条件表达式 |
| 数据定义 | create | 数据表的建立 | create table tablename(字段 1, 字段 2....) |
| | drop | 数据表的删除 | drop table tablename |
| 数据控制 | grant | 为用户授予系统权限 | grant<系统权限> <角色> [,<系统权限> <角色>]... to <用户名> <角色> public[,<用户名> <角色>]... [with admin option] |
| | revoke | 收回系统权限 | revoke <系统权限> <角色> [,<系统权限> <角色>]... from<用户名> <角色> public[,<用户名> <角色>]... |

例如，设教务管理系统中有 3 个基本表：

学生信息表 S(SNO, SNAME, AGE, SEX)，其属性分别表示学号、学生姓名、年龄和性别。

选课信息表 SC(SNO, CNO, SCGRADE)，其属性分别表示学号、课程号和成绩。

课程信息表 C(CNO, CNAME, CTEACHER)，其属性分别表示课程号、课程名称和任课老师姓名。

(1) 把 SC 表中每门课程的平均成绩插入到另外一个已经存在的表 SC_C(CNO, CNAME, AVG_GRADE)中，其中 AVG_GRADE 表示的是每门课程的平均成绩。

```
INSERT INTO SC_C(CNO, CNAME, AVG_GRADE)
SELECT SC.CNO, C.NAME, AVG(SCGRADE) FROM SC, C WHERE SC.CNO = C.CNO
```

(2) 从 SC 表中把何昊老师的女学生选课记录删除。

```
DELETE FROM SC, S, C WHERE SC.SNO = S.SNO AND SC.CNO=C.CNO AND C.CTEACHER='何昊'
```

(3) 规定女同学选修何昊老师的课程成绩都应该在 80 分以上（包含 80 分）。

```
ALTER TABLE SC, S, C
ADD CONSTRAINT GRADE CHECK(GRADE>=80)
WHERE SC.CNO=C.CNO and SC.SNO=S.SNO AND C.CTEACHER='何昊'
```

(4) 找出没有选修过“何昊”老师讲授课程的所有学生姓名。

```
SELECT SNAME FROM S
WHERE NOT EXISTS(
SELECT * FROM SC,C WHERE SC.CNO=C.CNO AND CNAME='何昊' AND SC.SNO=S.SNO)
```

(5) 列出有两门以上（含两门）不及格课程（成绩小于 60）的学生姓名及其平均成绩。

```
SELECT S.SNO,S.SNAME,AVG_SCGRADE=AVG(SC.SCGRADE)
FROM S,SC,(
SELECT SNO FROM SC WHERE SCGRADE<60 GROUP BY SNO
HAVING COUNT(DISTINCT CNO)>=2)A WHERE S.SNO=A.SNO AND SC.SNO=A.SNO
GROUP BY S.SNO,S.SNAME
```

(6) 列出既学过“1”号课程，又学过“2”号课程的所有学生姓名。

```
SELECT S.SNO,S.SNAME
FROM S,(SELECT SC.SNO FROM SC,C
WHERE SC.CNO=C.CNO AND C.CNAME IN('1','2')
GROUP BY SNO
HAVING COUNT(DISTINCT CNO)=2
)SC WHERE S.SNO=SC.SNO
```

(7) 列出“1”号课成绩比“2”号同学该门课成绩高的所有学生的学号。

```
SELECT S.SNO,S.SNAME
FROM S,(
SELECT SC1.SNO
```

```
FROM SC SC1,C C1,SC SC2,C C2
WHERE SC1.CNO=C1.CNO AND C1.NAME='1'
AND SC2.CNO=C2.CNO AND C2.NAME='2'
AND SC1.SCGRAD>SC2.SCGRAD
)SC WHERE S.SNO=SC.SNO
```

(8) 列出“1”号课成绩比“2”号课成绩高的所有学生的学号及其“1”号课和“2”号课的成绩。

```
SELECT S.SNO,S.SNAME,SC.[1 号课成绩],SC.[2 号课成绩]
FROM S,(
SELECT SC1.SNO,[1 号课成绩]=SC1.SCGRAD,[2 号课成绩]=SC2.SCGRAD
FROM SC SC1,C C1,SC SC2,C C2
WHERE SC1.CNO=C1.CNO AND C1.NAME='1'
AND SC2.CNO=C2.CNO AND C2.NAME='2'
AND SC1.SCGRAD>SC2.SCGRAD
)SC WHERE S.SNO=SC.SNO
```

8.1.3 内连接与外连接有什么区别

内连接也称为自然连接，只有两个表相匹配的行才能在结果集中出现。返回的结果集是两个表中所有相匹配的数据，而舍弃不匹配的数据。由于内连接是从结果表中删除与其他连接表中没有匹配行的所有行，所以内连接可能会造成信息的丢失。内连接的语法如下：

```
select fieldlist from table1 [inner] join table2 on table1.column=table2.column
```

内连接是保证两个表中所有的行都要满足连接条件，而外连接则不然。与内连接不同，外连接不仅包含符合连接条件的行，而且还包括左表（左外连接时）、右表（右外连接时）或两个边接表（全外连接）中的所有数据行。也就是说，只限制其中一个表的行，而不限制另一个表的行。SQL 的外连接共有 3 种类型：左外连接，关键字为 LEFT OUTER JOIN；右外连接，关键字为 RIGHT OUTER JOIN；全外连接，关键字为 FULL OUTER JOIN。外连接的用法和内连接一样，只是将 INNER JOIN 关键字替换为相应的外连接关键字即可。

内连接只显示符合连接条件的记录，外连接除了显示符合连接条件的记录外（如用左外连接），还显示左表中的记录。

表 8-2 所示为学生表 A，表 8-3 所示为学生表 B。

表 8-2 学生表 A

| 学号 | 姓名 |
|------|----|
| 0001 | 张三 |
| 0002 | 李四 |
| 0003 | 王五 |

表 8-3 学生表 B

| 学号 | 课程名 |
|------|-----|
| 0001 | 数学 |
| 0002 | 英语 |
| 0003 | 数学 |
| 0004 | 计算机 |

对表 A 和表 B 进行内连接后的结果见表 8-4。

对表 B 和表 A 进行左外连接后的结果见表 8-5。

表 8-4 内连接

| 学号 | 姓名 | 课程名 |
|------|----|-----|
| 0001 | 张三 | 数学 |
| 0002 | 李四 | 英语 |
| 0003 | 王五 | 数学 |

表 8-5 左外连接

| 学号 | 姓名 | 课程名 |
|------|----|-----|
| 0001 | 张三 | 数学 |
| 0002 | 李四 | 英语 |
| 0003 | 王五 | 数学 |
| 0004 | | 计算机 |

8.1.4 什么是事务

事务是数据库中一个单独的执行单元 (unit)，它通常由高级数据库操作语言 (如 SQL) 或编程语言 (如 C++、Java 等) 书写的用户程序的执行所引起。当在数据库中更改数据成功时，在事务中更改的数据便会提交，不再改变。否则，事务就取消或者回滚，更改无效。

例如，网上购物的交易过程至少包括以下几个步骤的操作：

- (1) 更新客户所购商品的库存信息。
- (2) 保存客户付款信息。
- (3) 生成订单并且保存到数据库中。
- (4) 更新用户相关信息，如购物数量等。

在正常的情况下，这些操作都将顺利进行，最终交易成功，与交易相关的所有数据库信息也成功地更新。但是，如果遇到突然掉电或是其他意外情况，导致这一系列过程中任何一个环节出了差错，如在更新商品库存信息时发生异常、顾客银行账户余额不足等，都将导致整个交易过程失败。而一旦交易失败，数据库中所有信息都必须保持交易前的状态不变，比如最后一步更新用户信息时失败而导致交易失败，那么必须保证这笔失败的交易不影响数据库的状态，即原有的库存信息没有被更新、用户也没有付款、订单也没有生成。否则，数据库的信息将会不一致，或者出现更为严重的不可预测的后果，数据库事务正是用来保证这种情况下交易的平稳性和可预测性的技术。

事务必须满足 4 个属性，即原子性 (atomicity)、一致性 (consistency)、隔离性 (isolation)、持久性 (durability)，即 ACID 4 种属性。

(1) 原子性。事务是一个不可分割的整体，为了保证事务的总体目标，事务必须具有原子性，即当数据修改时，要么全执行，要么全都不执行。即不允许事务部分地完成，避免了只执行这些操作的一部分而带来的错误。原子性要求事务必须被完整执行。

(2) 一致性。一个事务执行之前和执行之后数据库数据必须保持一致性状态。数据库的一致性状态应该满足模式所指定的约束，那么在完整执行该事务后数据库仍然处于一致性状态。为了维护所有数据的完整性，在关系型数据库中，所有的规则必须应用到事务的修改上。数据库的一致性状态由用户来负责，由并发控制机制实现。例如，银行转账，转账前后两个账户金额之和应保持不变。由并发操作带来的数据不一致性包括丢失数据修改、读“脏”数据、不可重复读和产生幽灵数据。

(3) 隔离性。隔离性也被称为独立性，当两个或多个事务并发执行时，为了保证数据的安全性，将一个事物内部的操作与事务的操作隔离起来，不被其他正在进行的事务看到。例如，对任何一对事务 T1、T2，对 T1 而言，T2 要么在 T1 开始之前已经结束，要么在 T1 完成之后再开始执行。数据库有 4 种类型的事务隔离级别：不提交的读、提交的读、可重复的读和串行化。因为隔离性使得每个事务的更新在它被提交之前，对其他事务都是不可见的，所以实施隔离性是解决临时更新与消除级联回滚问题的一种方式。

(4) 持久性。持久性也被称为永久性，事务完成以后，DBMS 保证它对数据库中的数据的修改是永久性的，当系统或介质发生故障时，该修改也永久保持。持久性一般通过数据库备份与恢复来保证。

严格来说，数据库事务属性 (ACID) 都是由数据库管理系统来进行保证的，在整个应用程序运行过程中应用无需去考虑数据库的 ACID 实现。

一般情况下，通过执行 COMMIT 或 ROLLBACK 语句来终止事务，当执行 COMMIT 语

句时，自从事务启动以来对数据库所做的一切更改就成为永久性的了，即被写入到磁盘；而当执行 ROLLBACK 语句时，自动事务启动以来对数据库所做的一切更改都会被撤销，并且数据库中的内容返回到事务开始之前所处的状态。无论什么情况，在事务完成时，都能保证回到一致状态。

8.1.5 什么是存储过程？它与函数有什么区别与联系

SQL 语句执行的时候要先编译，然后再被执行。在大型数据库系统中，为了提高效率，将为了完成特定功能的 SQL 语句集进行编译优化后，存储在数据库服务器中，用户通过指定存储过程的名字来调用执行。

创建存储过程的常用语法如下：

```
create procedure sp_name @[参数名][类型]
as
begin
...
end
```

调用存储过程语法：exec sp_name [参数名]。

删除存储过程语法：drop procedure sp_name。

使用存储过程可以增强 SQL 语言的功能和灵活性，由于可以用流程控制语句编写存储过程，有很强的灵活性，所以可以完成复杂的判断和运算，并且可以保证数据的安全性和完整性，同时存储过程可以使没有权限的用户在控制之下间接地存取数据库，也保证了数据的安全。

但存储过程不等于函数，两者虽然本质上没有区别，但具体而言有以下几个方面的区别：

(1) 存储过程一般是作为一个独立的部分来执行的，而函数可以作为查询语句的一个部分来调用。由于函数可以返回一个对象，因此它可以在查询语句中位于 From 关键字的后面。

(2) 一般而言，存储过程实现的功能较复杂，而函数实现的功能针对性比较强。

(3) 函数需要用括号包住输入的参数，且只能返回一个值或表对象，存储过程可以返回多个参数。

(4) 函数可以嵌入在 SQL 中使用，可以在 select 中调用，存储过程不行。

(5) 函数不能直接操作实体表，只能操作内建表。

(6) 存储过程在创建时即在服务器上进行了编译，执行速度更快。

8.1.6 什么是主键？什么是外键

主键也称为主码，是数据库中的一个或多个字段，是表中记录的唯一标示符。主键不能为空。一个表中只能有一个主键，主键列不一定只有一列，可以是多列。

例如，学生表(学号，姓名，性别，班级，学校)。

姓名可以有重复的，不能称为主键；而每个学生的学号是唯一的，学号就为一个主键。

由概念可知，主键可唯一地标识一行。另外，主键可作为一个可以被外键有效使用的对象，因此主键需遵循以下原则：

(1) 主键对用户而言没有意义。

(2) 主键不能为空。

(3) 主键保持不变。因为主键的用途是唯一地标识一行数据。

(4) 主键不应包含动态变化的时间戳。

(5) 主键原则上应当由计算机自动生成，而非用户指定。

外键也称外码，表示的是两个关系之间的联系。当公共关键字在一个关系中为主键时，这个公共关键字被称为另外一个关系的外键。假设有两个表 A、B，key 是 A 的主键，同时也是 B 中的字段，那么 key 称为 B 的外键。

例如：

学生表(学号，姓名，性别，班级，学校)

成绩表(学号，课程号，成绩)

学号为学生表中的主键，同时也是成绩表中的字段，所以学号为成绩表的外键。外键用来和其他表建立联系，实现表之间的关联。保持数据的一致性，实现参照完整性等约束。一个表可以有多个外键，也可以为空。

8.1.7 什么是死锁

在操作系统中有若干程序并发执行，它们不断地申请、释放资源，在此过程中，由于争夺资源而处于无限期的等待状态，造成程序无法继续执行，若无外力作用，它们都将无法推进下去，这时称系统处于死锁状态或系统产生了死锁。此时便只能通过外力来打破这种状态。

产生死锁的原因有以下 3 点。首先，系统资源不足，在系统中常常有多个进程共享资源的情况，如打印机，这些资源在同一时刻只能被一个进程使用。当资源数目不能满足进程时，便可能因为抢夺资源产生死锁。其次，进程运行推进顺序不对，进程在运行中具有异步性，当进程推进顺序不当时，便产生死锁。例如，进程 P1 和 P2，两进程同时具有 R1 和 R2 两个资源时，才能执行，当两进程并发执行时，若 P1 保持资源 R1，P2 保持资源 R2，双方都在等待对方释放资源，此时便发生了死锁。最后，资源分配不当，如果系统资源充足，进程的资源请求都能得到满足，死锁的可能性会被大大降低，而进程推进顺序与速度不同，也可能产生死锁。

总的来说，产生死锁有 4 个必要条件：1) 互斥，每个资源每次只能被一个进程使用；2) 请求与保持等待，一个进程因请求资源而被阻塞时，对已获得的资源保持不放；3) 不可剥夺，进程已获得的资源，在未使用完之前，不能强制剥夺；4) 环路等待，若干进程之间形成首尾相接的等待资源关系。

所以，为了预防死锁，就要打破产生死锁的 4 个条件中的一个或多个，因此需要最大限度地增加系统资源，合理地安排进程的顺序并确定合理的分配资源的算法。

避免死锁是在资源的动态分配过程中，采取有效的方法防止系统进入不安全状态，达到预防死锁的目的，其中最具代表性的方法就是银行家算法。

8.1.8 什么是共享锁？什么是互斥锁

在数据库中，锁主要是对数据进行读/写的一种保护机制，从数据库系统的角度来看，一般可以将锁分为共享锁和互斥锁。

共享锁简称 S 锁，也叫读锁。用于不更改或不更新数据的操作（只读操作），如 select 语句。如果事务 T 对数据 A 加上共享锁后，则其他事务只能对 A 再加共享锁，不能加排他锁。共享锁可阻止其他并发运行的程序获取重叠的独占锁定，但是允许该程序获取重叠的共享锁定。其他用户可以获取共享锁锁定的资源，但是不能进行修改该共享锁。在执行 select 命令时，sql server 通常会对对象进行共享锁锁定。若事务 T 对数据 D 加 S 锁，则其他事务只能对 D 加 S 锁，而不能加 X 锁，直至 T 释放 D 上的 S 锁；一般要求在读取数据前要向该数据加共享锁，所以共享锁又称为读锁。通常加共享锁的数据页被读取完毕后，共享锁就会立即被释放。

互斥锁简称 X 锁，也叫排他锁，用于数据修改操作，如 insert、update 或 delete。确保不会同时对同一资源进行多重更新。为了保证数据操作的完整性，引入了互斥锁。用互斥锁来保证在任意时刻，只能有一个线程访问对象。若事务 T 对数据 D 加 X 锁，则其他任何事务都不能再对 D 加任何类型的锁，直至 T 释放 D 上的 X 锁；一般要求在修改数据前要向该数据加排他锁，所以排他锁又称为写锁。

而对于锁的使用，也有一定的限制，需要遵守两个事项：1) 先锁后操作；2) 事务结束之后必须解锁。

8.1.9 一、二、三、四范式有何区别

在设计与操作维护数据库时，最关键的问题就是要确保数据正确地分布到数据库的表中，使用正确的数据结构，不仅有助于对数据库进行相应的存取操作，还可以极大地简化应用程序的其他内容（查询、窗体、报表、代码等）。正确地进行表的设计称为“数据库规范化”，它的目的就是减少数据库中的数据冗余，从而增加数据的一致性。

范化是在识别数据库中的数据元素、关系，以及定义所需的表和各表中的项目这些初始工作之后的一个细化的过程。常见的范式有 1NF、2NF、3NF、BCNF 以及 4NF。

1NF，第一范式。第一范式是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多值，即实体中的某个属性不能有多值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。第一范式的模式要求属性值不可再分裂成更小部分，即属性项不能是属性组合或由组属性组成。简而言之，第一范式就是无重复的列。例如，由“职工号”、“姓名”、“电话号码”组成的表（一个人可能有一个办公电话和一个移动电话），这时将其规范化为 1NF 可以将电话号码分为“办公电话”和移动电话两个属性，即职工（职工号，姓名，办公电话，移动电话）。

2NF，第二范式。第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或行必须可以被唯一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。如果关系模式 R 为第一范式，并且 R 中每一个非主属性完全函数依赖于 R 的某个候选键，则称 R 为第二范式模式。如果 A 是关系模式 R 的候选键的一个属性，则称 A 是 R 的主属性，否则称 A 是 R 的非主属性。例如，在选课关系表（学号，课程号，成绩，学分）中，关键字为组合关键字（学号，课程号），但由于非主属性学分仅依赖于课程号，对关键字（学号，课程号）只是部分依赖，而不是完全依赖，所以此种方式会导致数据冗余以及更新异常等问题，解决办法是将其分为两个关系模式：学生表（学号，课程号，分数）和课程表（课程号，学分）。新关系通过学生表中的外关键字课程号联系，在需要时进行连接。

3NF，第三范式。如果关系模式 R 是第二范式，且每个非主属性都不传递依赖于 R 的候选键，则称 R 是第三范式的模式。例如，学生表（学号，姓名，课程号，成绩），其中学生姓名无重名，所以该表有两个候选码（学号，课程号）和（姓名，课程号），则存在函数依赖：学号→姓名，（学号，课程号）→成绩，（姓名，课程号）→成绩，唯一的非主属性成绩对码不存在部分依赖，也不存在传递依赖，所以属于第三范式。

BCNF。它构建在第三范式的基础上，如果关系模式 R 是第一范式，且每个属性都不传递依赖于 R 的候选键，那么称 R 为 BCNF 的模式。假设仓库管理关系表（仓库号，存储物品号，管理员号，数量），满足一个管理员只在一个仓库工作；一个仓库可以存储多种物品。则

存在如下关系：

(仓库号, 存储物品号) → (管理员号, 数量)

(管理员号, 存储物品号) → (仓库号, 数量)

所以, (仓库号, 存储物品号) 和 (管理员号, 存储物品号) 都是仓库管理关系表的候选码, 表中的唯一非关键字段为数量, 它是符合第三范式的。但是, 由于存在如下决定关系:

(仓库号) → (管理员号)

(管理员号) → (仓库号)

即存在关键字段决定关键字段的情况, 所以其不符合 BCNF 范式。把仓库管理关系表分解为两个关系表: 仓库管理表 (仓库号, 管理员号) 和仓库表 (仓库号, 存储物品号, 数量), 这样的数据库表是符合 BCNF 范式的, 消除了删除异常、插入异常和更新异常。

4NF, 第四范式。设 R 是一个关系模式, D 是 R 上的多值依赖集合。如果 D 中成立非平凡多值依赖 $X \twoheadrightarrow Y$ 时, X 必是 R 的超键, 那么称 R 是第四范式的模式。例如, 职工表 (职工编号, 职工孩子姓名, 职工选修课程), 在这个表中同一个职工也可能会有多个职工孩子姓名, 同样, 同一个职工也可能会有多个职工选修课程, 即这里存在着多值事实, 不符合第四范式。如果要符合第四范式, 只需要将上表分为两个表, 使它们只有一个多值事实。例如, 职工表一 (职工编号, 职工孩子姓名), 职工表二 (职工编号, 职工选修课程), 两个表都只有一个多值事实, 所以符合第四范式。

图 8-1 所示为各范式关系图。

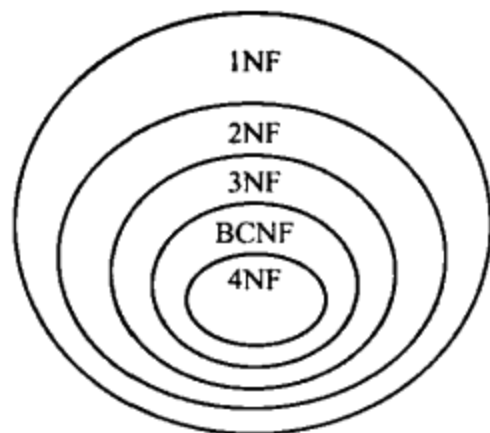


图 8-1 各范式关系图

8.1.10 如何取出表中指定区间的记录

写一个 SQL 语句, 取出表 S 中第 21~30 记录 (SQL Server, 以自动增长的 ID 作为主键, ID 可能不连续)

方法一: `Select Top 10 * From S Where ID > (Select MAX(ID) From (Select Top 20 ID From S) as S)`

方法二: `Select Top 10 * From S Where ID NOT IN (Select Top 20 ID From S)`

8.1.11 什么是 CHECK 约束

CHECK 约束是指限制表中某一列或某些列中可接受的数据值或数据格式, 它用于限制列的取值范围, 使用形式为: CHECK (约束表达式)。如果是对单列定义 CHECK 约束, 那么该列只允许特定的值; 如果是对一个表定义 CHECK 约束, 那么此约束会在特定的列中对值进行限制。例如, 对于一个员工信息表, 给员工的年龄属性添加了一个约束, 即年龄必须大于 0 且小于等于 120, 那么用户在输入年龄的时候, 就必须遵守该约束, 输入负数或者 121 就无法输入。

8.1.12 什么是视图

视图是由从数据库的基本表中选取出来的数据组成的逻辑窗口, 不同于基本表。它是一个虚表, 在数据库中, 存放的只是视图的定义而已, 不存放视图包含的数据项, 这些项目仍然存放在原来的基本表结构中。

视图的作用非常多, 主要有以下几点: 首先可以简化数据查询语句; 其次可以使用户能从多角度看待同一数据; 然后, 通过引入视图, 可以提高数据的安全性; 最后, 视图提供了一定程度的逻辑独立性等。

通过引入视图机制, 用户可以将注意力集中在其关心的数据上而非全部数据, 这样就大大

提高了用户效率与用户满意度，而且如果这些数据来源于多个基本表结构，或者数据不仅来自于基本表结构，还有一部分数据来源于其他视图，并且搜索条件又比较复杂时，需要编写的查询语句就会比较繁琐，此时定义视图就可以使数据的查询语句变得简单可行。定义视图可以将表与表之间复杂的操作连接和搜索条件对用户不可见，用户只需要简单地对一个视图进行查询即可，所以增加了数据的安全性，但是不能提高查询的效率。

8.2 SQL 高级应用

对于实际的应用系统而言，基本的数据库操作根本无法满足实际的需求，随着数据表的增多、数据量的增大，数据库的效率以及安全性问题就会变得日益突出，数据库中引入了触发器、游标、索引等内容来满足这些需求。

8.2.1 什么是触发器

触发器是一种特殊类型的存储过程，它由事件触发，而不是程序调用或手工启动。当数据库有特殊的操作时，对这些操作由数据库中的事件来触发，自动完成这些 SQL 语句。使用触发器可以用来保证数据的有效性和完整性，完成比约束更复杂的数据约束。

触发器与存储过程的区别见表 8-6。

表 8-6 触发器与存储过程的区别

| 触 发 器 | 存 储 过 程 |
|-------------------------------|--|
| 当某类数据操纵 DML 语句发生时隐式地调用 | 从一个应用或过程中显式地调用 |
| 在触发器体内禁止使用 COMMIT、ROLLBACK 语句 | 在过程体内可以使用所有 PL/SQL 块中都能使用的 SQL 语句，包括 COMMIT、ROLLBACK |
| 不能接受参数输入 | 可以接受参数输入 |

根据 SQL 语句的不同，触发器可分为两类：DML 触发器和 DLL 触发器。

DML 触发器是当数据库服务器发生数据操作语言事件时执行的存储过程，有 After 和 Instead Of 两种触发器。After 触发器被激活触发是在记录改变之后进行的一种触发器。Instead Of 触发器是在记录变更之前，去执行触发器本身所定义的操作，而不是执行原来 SQL 语句里的操作。DLL 触发器是在响应数据定义语言事件时执行的存储过程。

触发器的主要作用表现在以下几个方面：

- (1) 增加安全性。
- (2) 利用触发器记录所进行的修改以及相关信息，跟踪用户对数据库的操作，实现审计。
- (3) 维护那些通过创建表时的声明约束不可能实现的复杂的完整性约束以及对数据库中特定事件进行监控与响应。
- (4) 实现复杂的非标准的数据库相关完整性规则、同步实时地复制表中的数据。
- (5) 触发器是自动的，它们在对表的数据做了任何修改之后就会被激活。例如，可以自动计算数据值，如果数据的值达到了一定的要求，则进行特定的处理。以某企业财务管理为例，如果企业的资金链出现短缺，并且达到某种程度时，则发送警告信息。

下面是一个触发器的例子，该触发器的功能是在每周末进行数据表更新，如果当前用户没有访问 WEEKEND_UPDATE_OK 表的权限，需要重新赋予权限。

```
CREATE OR REPLACE TRIGGER update_on_weekends_check
```



```
BEFORE UPDATE OF sal ON EMP
FOR EACH ROW
DECLARE
my_count number(4);
BEGIN
SELECT COUNT(u_name)
FROM WEEKEND_UPDATE_OK INTO my_count
WHERE u_name = user_name;
IF my_count=0 THEN
RAISE_APPLICATION_ERROR(20508, 'Update not allowed');
END IF;
END;
```

引申：触发器分为事前触发和事后触发，两者有什么区别？语句级触发和行级触发有什么区别？

事前触发发生在事件发生之前验证一些条件或进行有一些准备工作；事后触发发生在事件发生之后，做收尾工作，保证事务的完整性。而事前触发可以获得之前和新的字段值。语句级触发器可以在语句执行之前或之后执行，而行级触发在触发器所影响的每一行触发一次。

8.2.2 什么是索引

索引是一种提高数据库查询速度的机制，它是一个在数据库的表或视图上按照某个关键字段的值，升序或降序排序创建的对象。当用户查询索引字段时，它可以快速地执行检索操作，借助索引，在执行查询的时候不需要扫描整个表就可以快速地找到所需要的数据。索引是与表或视图关联的磁盘上结构，即对表中列值排序的一种结构，可以加快从表或视图中检索行的速度，执行查询时不必扫描整个表就能更加快速地访问数据库中的信息。

用 CREATE INDEX 命令创建索引如下：

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED]
INDEX index_name ON {table | view} column [ ASC | DESC ] [,...n] [WITH
    [PAD_INDEX]
    [[,] FILLFACTOR = fillfactor]
    [[,] IGNORE_DUP_KEY]
    [[,] DROP_EXISTING]
    [[,] STATISTICS_NORECOMPUTE]
    [[,] SORT_IN_TEMPDB] ]
    [ON filegroup]
```

例如，有学生信息表 student，内容见表 8-7。

可以对学校建立一个索引，见表 8-8。

表 8-7 学生信息表 student

| 学号 | 姓名 | 性别 | 年龄 |
|-----|----|----|----|
| 101 | 张三 | 男 | 19 |
| 201 | 李四 | 女 | 20 |
| 102 | 王五 | 男 | 19 |
| 202 | 赵六 | 男 | 18 |
| 103 | 田七 | 女 | 18 |

表 8-8 索引

| 学号 | 记录号 |
|-----|-----|
| 101 | 1 |
| 102 | 3 |
| 103 | 5 |
| 201 | 2 |
| 202 | 4 |

例如，为表“商品”基于“货号”字段创建一个唯一的簇索引代码：

```
create unique clustered index PK_商品
on 商品(货号)
```



```
with  
pad_index,  
fillfactor=10,  
drop_existing
```

一条索引记录包含键值和逻辑指针。创建索引时，系统分配一个索引页。在表中插入一行数据，同时也向该索引页中插入一行索引记录。索引记录包含的索引字段值比真实数据量小，节省了空间。

索引的类型有聚焦索引和非聚焦索引。聚焦索引是表中的行的物理顺序与键值的逻辑顺序一样，一个表只能有一个聚焦索引。与非聚焦索引相比，聚焦索引一般情况下可以获得更快的数据访问速度。非聚焦索引是数据存储与索引存储不在同一个地方。索引中有指针，该指针指向数据的存储位置，索引中的项目按索引之前的顺序存储，而表中的信息按另一种顺序存储。

创建索引可以大大提高系统的性能，主要表现在以下几个方面：1) 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。2) 通过索引，可以大大加快数据的检索速度。3) 通过索引可以加速表与表之间的连接，从而有效实现数据的参考完整性。4) 在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间。5) 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

索引可以有效地提高查询效率，那为什么不将所有的列都建立索引呢？其实索引尽管可以带来方便，但并非越多越好，过多的索引也会带来许多不利的问题。首先，创建索引和维护索引要耗费时间、空间。当数据量比较小时，这种问题还不够突出；而当数据量比较大时，这种缺陷会比较明显，效率会非常低下。其次，除了数据表占数据空间之外，每一个索引还需要占用一定的物理空间。如果要建立聚簇索引，那么需要的空间就会更大，从而造成不必要的空间浪费。最后，当对表中的数据进行增加、删除和修改的时候，索引也要动态地维护，从而降低了数据的维护速度。

8.2.3 什么是回滚

为了保证在应用程序、数据库或系统出现错误后，数据库能够被还原，以保证数据库的完整性，所以需要进行回滚。回滚（rollback）就是在事务提交之前将数据库数据恢复到事务修改之前数据库数据状态。

回滚执行相反的操作，可以撤销错误的操作，从而保证数据的完整性。例如，用户 A 给用户 B 转账，在数据库中就需要给 A 与 B 的账户信息进行修改（update）操作，而这两条 sql 语句必须都执行或者都不执行。例如，先执行用户 B 的修改（update）语句，使用户 B 的账户金额增加了 1000，然后执行用户 A 的 update 语句，使用户 A 的账户金额减少 1000。如果用户 A 的账户余额大于 1000，则交易顺利进行，不存在任何问题，但是当用户 A 的账户余额小于 1000 时，由于转账金额不允许大于账户余额，第二条 sql 语句就无法正确执行，此时，数据库的状态必须回到没有执行 B 的 update 语句之前，需要进行回滚操作，回滚就是执行一遍相反的操作，此时再执行 B 的 update 金额减 1000。

需要注意回滚与撤销的区别。回滚是指将数据库的状态恢复到执行事务之前的状态，其中可能会使用 UNDO 日志进行回滚。撤销是一种记录日志的方式，并不是主要服务于事务回滚，而是主要用于系统从故障中恢复。例如，系统突然断电，系统要根据 UNDO 日志对未完成的事务进行处理，保证数据库的状态为执行这些事务前的状态。

8.2.4 数据备份有哪些种类

在网络运行与维护过程中,经常有一些难以预料的因素会导致数据的丢失,如硬件损坏、操作失误等,而且丢失的数据通常又会对企业的业务产生非常不利的影响,所以必须不定期地对数据进行及时备份,以便在灾难发生后能够迅速地恢复数据。数据备份就是保存数据的备份,目的是为了预防灾难造成的数据损失。它一般分为完全备份、差异备份、事务日志备份、增量备份几大类。

完全备份是将数据库中的全部信息进行备份,它是恢复的基线,在进行完全备份时,不但备份数据库的数据文件、日志文件,还需要备份文件的存储位置信息以及数据库中的全部对象和相关信息。在对数据库进行完全备份时,所有未完成的事务或发生在备份过程中的事务都将被忽略,如果使用完全数据库备份类型,那么从开始备份到开始恢复这段时间内发生的任何针对数据库的修改都将无法恢复。所以,只有在一定的要求或条件下才使用这种备份类型。

差异备份是备份从最近的完全备份之后对数据所作的修改,它以完全备份为基准点,备份完全备份之后变化了的数据文件、日志文件以及数据库中其他被修改的内容。差异备份耗费的时间比完全备份少,但也会占用一些时间,同完全备份一样,差异备份过程中也允许用户访问数据库并对数据进行操作,并且在差异备份过程中会把这些操作也一起备份起来。

事务日志备份是备份从上次备份之后的日志记录,而且在默认情况下,事务日志备份完成后要截断日志,事务日志备份记录了用户对数据进行的修改操作。随着时间的推移,日志中的记录数会越来越多,容量有时比数据库备份大,这样势必会占满整个磁盘空间。因此,为了避免这种情况的发生,必须定期地将日志记录中不必要的记录清除掉,以节省空间。清除掉无用日志记录的过程叫做截断日志。

增量备份是针对上次备份的,备份上一次备份后所有发生变化的文件。在增量备份过程中,只备份有标记的选中的文件和文件夹,它清除标记,即备份后标记文件。

例如,对于数据库而言,按照生活规律,一般应该在星期一进行完全备份,在星期二至星期五进行差异备份。如果在星期五数据被破坏了,则只需要还原星期一完全的备份和星期四的差异备份。这种策略备份数据需要较多的时间,但还原数据使用较少的时间。

再例如,在星期一进行完全备份,在星期二至星期五进行增量备份。如果在星期五数据被破坏了,则需要还原星期一正常的备份和从星期二至星期五的所有增量备份。这种策略备份数据需要较多的时间,但还原数据使用较少的时间。

与数据备份相对应的就是数据恢复,数据恢复是指将数据恢复到事故之前的状态,可以将其看成是数据备份操作的逆过程。数据备份是数据恢复的前提,数据恢复是数据备份的目的,无法恢复的数据备份是没有任何意义的。

8.2.5 什么是游标

在数据库中,游标提供了一种对从表中检索出的数据进行操作的灵活手段。它实际上是一种能从包括多条数据记录的结果集中每次提取一条记录的机制。

游标总是与一条 SQL 选择语句相关联,因为游标由结果集(可以是零条、一条或由相关的选择语句检索出的多条记录)和结果集中指向特定记录的游标位置组成。当决定对结果集进行处理时,必须声明一个指向该结果集的游标。

游标允许应用程序对查询语句 `select` 返回的行结果集中每一行进行相同或不同的操作,而不是一次对整个结果集进行同一种操作。它还提供对基于游标位置而对表中数据进行删除或

更新的能力。而且，正是游标把作为面向集合的数据库管理系统和面向行的程序设计两者联系起来，使两个数据处理方式能够进行沟通。

例如，声明一个游标 `student_cursor`，用于访问数据库 `SCHOOL` 中的“学生基本信息表”，代码如下：

```
USE SCHOOL
GO
DECLARE student_cursor CURSOR
FROM SELECT * FROM 学生基本信息表
```

上述代码中，声明游标时，在 `SELECT` 语句中未使用 `WHERE` 子句，故此游标返回的结果集是由“学生基本信息表”中的所有记录构成的。

在 `select` 返回的行集合中，游标不允许程序对整个行集合执行相同的操作，但对每一行数据的操作不作要求。游标的优点有以下两个方面的内容：

- (1) 在使用游标的表中，对行提供删除和更新的能力。
- (2) 游标将面向集合的数据库管理系统和面向行的程序设计连接了起来。

8.2.6 并发环境下如何保证数据的一致性

并发一般是指多用户同时访问相同的数据。数据一致性是指系统中每个用户都能够取得具备一致性的数据，同时还能够看到自己或其他用户所提交的事务对数据的修改。

在并发环境下，一般可以采用多种机制来保证数据的一致性。例如，Oracle 系统提供的事务级的一致性、行级锁、表级锁等。下面只介绍行级锁。

提交读和串行性事务都使用行级锁，都会在试图修改一个没有提交的并行事务更新的行时产生等待。第二个事务等待其他事务提交或者撤销来释放它的锁，才能更新给定的行。不管等待的是什么级别的隔离事务，若其他事务回滚了，都可以对以前锁定的行进行修改。

但是如果其他事务提交并释放了它的锁，提交事务可执行它想要的修改。

8.2.7 如果数据库日志满了，会出现什么情况

日志文件 (Log File) 记录所有对数据库数据的修改，主要是保护数据库以防止故障，以及恢复数据时使用。其特点如下：

- (1) 每一个数据库至少包含两个日志文件组。每个日志文件组至少包含两个日志文件成员。
- (2) 日志文件组以循环方式进行写操作。
- (3) 每一个日志文件成员对应一个物理文件。

通过日志文件来记录数据库事务可以最大限度地保证数据的一致性与安全性，但一旦数据库中日志满了，就只能执行查询等读操作，不能执行更改、备份等操作。其原因是任何写操作都要记录日志，也就是说基本上处于不能使用的状态。

8.2.8 如何判断谁往数据库中插入了一行数据

可以通过以下 3 种方法来达到这个目的：事先打开审计功能、表上建立触发器或者查看 `logmnr` 等。

- (1) 审计功能。

先设置 `audit_trail` 参数，决定审计结果的保存地点，然后执行 `audit insert on schema. table_name whenever successful`；当有执行 `insert` 操作的动作后，根据 `audit_trail` 参数到相应位置去看审计结果即可。

(2) 触发器。

```
create or replace trigger tname
after insert
on tname →判断此表是否被插入记录
for each row
begin
insert into ta(日期) values(sysdate);
commit;
end;
```

只需要将代码中的 **tname** 换成实际的表的名称即可。需要注意的是，建立触发器实质上也是审计，只是它是基于值的审计，比数据库审计慢。

(3) 通过 logmnr 日志查看。

logmnr 是 Oracle 公司提供的分析工具，该工具轻巧实用，使用该工具可以轻松获得 Oracle 重作日志文件（归档日志文件）中的具体内容，而且该工具可以分析出所有对于数据库操作的 DML（insert、update、delete 等）语句、DDL 语句等，另外还可分析得到一些必要的回滚语句。所以，该工具特别适用于调试、审计或者回滚某个特定的事务。

计算机网络技术是互联网发展的基础。它是计算机技术与通信技术结合的产物，是现在信息技术的一个重要组成部分，而且正朝着数字化、高速化、智能化的方向迅速发展。随着3G、4G 技术的兴起，越来越多的企业参与到了网络与通信相关的行业的角逐，网络与通信成为信息化浪潮的先锋。而对于网络相关技术的考察也越来越受到各大 IT 企业的重视。

9.1 网络模型

9.1.1 OSI 七层模型是什么

OSI（Open System Interconnection，开放系统互连）七层网络模型称为开放式网络互连参考模型。它是国际标准组织制定的一个指导信息互联、互通和协作的网络规范。开放是指只要遵循 OSI 标准，位于世界上任何地方的任何系统之间都可以进行通信，开放系统是指遵循互联协议的实际系统，如电话系统。从逻辑上可以将其划分为七层模型，由下至上分别为物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。其中，上三层称为高层，用于定义应用程序之间的通信和人机界面；下四层称为底层，用于定义数据如何进行端到端的传输（end-to-end），物理规范以及数据与光电信号间的转换。图 9-1 所示为其分层示例图。

| |
|-------------------|
| 应用层（Application） |
| 表示层（Presentation） |
| 会话层（Session） |
| 传输层（Transport） |
| 网络层（Network） |
| 数据链路层（Data Link） |
| 物理层（Physical） |

图 9-1 分层示例图

具体而言，从上往下每一层的功能如下：

- （1）应用层。应用层也称为应用实体，一般是指应用程序，该层主要负责确定通信对象，并确保有足够的资源用于通信。常见的应用层协议有 FTP、HTTP、SNMP 等。
- （2）表示层。表示层一般负责数据的编码以及转化，确保应用层能够正常工作。该层是界面与二进制代码间互相转化的地方，同时该层负责进行数据的压缩、解压，加密、解密等，该层也可以根据不同的应用目的将数据处理为不同的格式，表现出来就是各种各样的文件扩展名。
- （3）会话层。会话层主要负责在网络中的两个结点之间建立、维护、控制会话，区分不同的会话，以及提供单工（Simplex）、半双工（Half duplex）、全双工（Full duplex）3 种通信模式的服务。NFS、RPC、X Windows 等都工作在该层。
- （4）传输层。传输层是 OSI 模型中最重要的一层，它主要负责分割、组合数据，实现端到端的逻辑连接。数据在上三层是整体的，到了这一层开始被分割，这一层分割后的数据被称为段（Segment）。三次握手（Three-way handshake）、面向连接（Connection-Oriented）或非面向连接（Connectionless-Oriented）的服务、流量控制（Flow control）等都发生在这一层。工作在传输层的一种服务是 TCP/IP 中的 TCP（传输控制协议），另一项传输层服务是 IPX/SPX 协议集的 SPX（序列包交换）。常见的传输层协议有 TCP、UDP、SPX 等。
- （5）网络层。网络层是将网络地址翻译为物理地址，并决定将数据从发送方路由到接收

方,主要负责管理网络地址、定位设备、决定路由,路由器就工作在该层。上层的数据段在这一层被分割,封装后叫做包(Packet)。包有两种:一种为用户数据包(Data packets),是上层传下来的用户数据;另一种为路由更新包(Route update packets),是直接由路由器发出来的,用来和其他路由器进行路由信息的交换。常见的网络层协议有IP、RIP、OSPF等。

(6) 数据链路层。数据链路层为OSI模型的第二层,控制物理层与网络层之间的通信,主要负责物理传输的准备,包括物理地址寻址、CRC校验、错误通知、网络拓扑、流量控制、重发等。MAC地址和交换机都工作在这一层。上层传下来的包在这一层被分割封装后叫做帧(Frame)。常见的数据链路层协议有SDLC、STP、帧中继、HDLC等。

(7) 物理层。物理层是实实在在的物理链路,它规定了激活、维持、关闭通信端点之间的机械特性、电气特性、功能特性以及过程特性。它为上层协议提供了一个传输数据的物理媒体,负责将数据以比特流的方式发送、接收。常见的物理媒体有双绞线、同轴电缆等。属于物理层相关的规范有EIA/TIA RS-232、EIA/TIA RS-449、RJ-45等。

9.1.2 TCP/IP 模型是什么

TCP/IP (Transmission Control Protocol/Internet Protocol, 传输控制协议/因特网互联协议)是最基本的Internet协议,由网络层的IP与传输层的TCP构成。现在人们常提到的TCP/IP并不一定是指TCP和IP两个具体的协议,而是指的TCP/IP协议簇。

TCP/IP定义了电子设备如何连入Internet,以及数据如何在它们之间传输的标准。它基于四层参考模型,分别是网络接口层、网际层、传输层、应用层,每一层都呼叫它的下一层所提供的网络来完成自己的需求。

其中网络接口层负责底层的传输,常见的协议有Ethernet 802.3、Token Ring 802.5、X.25、HDLC、PPP、ATM等。网络层负责不同计算机之间的通信,一般包括IP、ICMP等内容。传输层提供应用程序间的通信,主要包括格式化信息流、提供可靠传输等。应用层用于向用户提供应用服务,如电子邮件、远程登录等。应用层协议一般有FTP、TELNET、SMTP等。属于TCP/IP协议簇的所有协议都位于该模型的上面三层。

TCP/IP并不完全符合OSI七层模型,它的每一层都对应于OSI七层模型中的一层或多层,图9-2所示是TCP/IP四层模型和OSI七层模型对应图。

| OSI 七层网络模型 | TCP/IP 四层模型 | 对应网络协议 |
|--------------------|-------------|---|
| 应用层 (Application) | 应用层 | TFTP, FTP, NFS, WAIS |
| 表示层 (Presentation) | | Telnet, Rlogin, SNMP, Gopher |
| 会话层 (Session) | 传输层 | SMTP, DNS |
| 传输层 (Transport) | | TCP, UDP |
| 网络层 (Network) | 网际层 | IP, ICMP, ARP, RARP, AKP, UUCP |
| 数据链路层 (Data Link) | 网络接口层 | FDDI, Ethernet, Arpanet, PDN, SLIP, PPP |
| 物理层 (Physical) | | IEEE 802.1A, IEEE 802.2 到 IEEE 802.11 |

图 9-2 TCP/IP 四层模型和 OSI 七层模型对应图

9.1.3 B/S 与 C/S 有什么区别

C/S 是 Client/Server (客户端/服务器) 的缩写,在 C/S 架构中,服务器通常采用高性能的

PC、工作站或者小型机，而且采用大型数据库系统，如 SQL Server、DB2、Oracle 或 Sybase 等。客户端需要安装专用的客户端软件。

B/S 是 Brower/Server（浏览器/服务器）的缩写，客户端通常只需要安装一个浏览器（Browser），如 Firefox、IE、Chrome 等即可，服务器安装 SQL Server、DB2、Oracle 或 Sybase 等数据库。在 B/S 架构中，用户界面完全通过浏览器实现，一部分事务逻辑在前端实现，主要事务逻辑在服务器端实现。浏览器通过 Web 服务器同数据库进行数据交互。

具体而言，两种设计结构存在以下几个方面的区别：

（1）硬件要求不同。C/S 一般建立在专用的网络上，是小范围的网络环境；而 B/S 一般构建于广域网之上，不需要专门的网络硬件环境，只要能接入网络即可。在 B/S 架构的应用中，客户端只需要能够运行浏览器就可以了。

（2）架构要求不同。C/S 程序更加注重流程，需要对权限多层次校验，对系统运行速度可以较少考虑。而 B/S 对安全以及访问速度需要多重的考虑，建立在需要更加优化的基础之上，比 C/S 有更高的要求。

（3）安全要求不同。C/S 一般面向相对固定的用户群，对信息安全的控制能力很强。一般高度机密的信息系统适宜采用 C/S 结构，可以通过 B/S 发布部分可以公开的信息。B/S 构建在广域网之上，对安全的控制能力相对弱，可能面向不可知的用户。

（4）系统维护不同。C/S 程序由于整体性导致升级比较困难，可能需要重做一个全新的系统，而 B/S 基于构件组成，只需要进行构建局部的更换就可以实现系统的无缝升级，将系统维护开销减到最小，用户从网上自己下载安装就可以实现升级。

（5）软件的重用性不同。因为整体性考虑，C/S 程序中构件的重用性不如在 B/S 架构下的构件的重用性好。因为 B/S 的多重结构，要求构件相对独立的功能，能够相对较好的重用，而 C/S 则很难做到这一点。

（6）用户接口不同。C/S 多是建立在操作系统平台上，表现方法有限，而 B/S 建立在浏览器上，有更加丰富和生动的表现方式与用户交流，并且大部分难度小，成本低。

9.1.4 MVC 模型结构是什么

MVC 是模型（Model）、视图（View）和控制（Controller）这 3 个单词的第一个字母。它是一种目前广泛流行的应用模型。它的目的是实现 Web 系统的职能分工。图 9-3 所示为 MVC 模型关系图。其中，模型层实现系统中的业务逻辑，通常可以用 JavaBean 或 EJB 来实现；视图层则用于与用户的交互，通常用 JSP 来实现；控制层则是模型与视图 View 之间沟通的桥梁，它可以把用户的请求分派并选择恰当的视图来显示它们，同时它也可以解释用户的输入并将其映射为模型层能够执行的操作。

MVC 强制性地分离 Web 应用的输入、处理和输出，使得 MVC 应用程序被分成 3

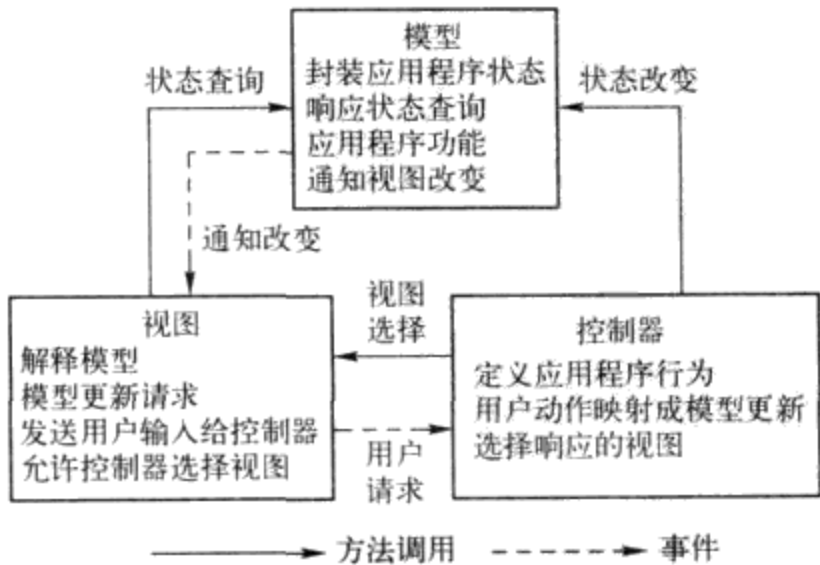


图 9-3 MVC 模型关系图

个核心部件：模型、视图、控制器。它们各自处理自己的任务。

(1) 模型（业务逻辑层）。

模型表示企业数据和业务逻辑，它是应用程序的主体部分。业务流程的处理过程对其他层来说是黑箱操作，模型接受视图请求数据，并返回最终的处理结果。业务模型的设计可以说是 MVC 最主要的核心。目前流行的 EJB 模型就是一个典型的应用例子。它从应用技术实现的角度对模型做了进一步的划分，以便充分利用现有的组件，但它不能作为应用设计模型的框架。它仅仅告诉设计人员按这种模型设计就可以利用某些技术组件，从而减少了技术上的困难，可以专注于业务模型的设计。

MVC 把应用的模型按一定的规则抽取出来，抽取的层次很重要，这也是判断设计人员是否优秀的主要依据。抽象与具体不能隔得太远，也不能太近。MVC 并没有提供模型的设计方法，而只告诉设计人员应该如何组织管理这些模型，以便于模型的重构和提高重用性。

业务模型还有一个很重要的模型那就是数据模型。数据模型主要指实体对象的数据持续化。例如，将一张订单保存到数据库，从数据库获取订单，将这个模型单独列出，所有有关数据库的操作只限制在该模型中。

(2) 视图（表示层）。

视图是用户看到并与之交互的界面。对早期的 Web 应用来说，视图就是由 HTML 元素组成的界面。在新式的 Web 应用中，HTML 依旧在视图中扮演着重要的角色，但一些新的技术已层出不穷，它们包括 Adobe Flash 和 XHTML，XML/XSL 等一些标识语言和 Web 服务等。

随着 Web 应用开发技术的发展，用户要求的日益提高，如何处理应用程序的界面已经变得越来越有挑战性。MVC 架构一个大的好处是它能为 Web 应用处理很多不同的视图。在视图中其实没有真正的业务处理发生，不管这些数据是联机存储的还是一个雇员列表，作为视图来讲，它只是作为一种输出数据并允许用户操纵的方式。

视图功能强大，主要表现在以下几个方面：

- 1) 根据客户类型显示信息。
- 2) 显示商业逻辑（模型）的结构，而不关心信息如何获得何时获得。

(3) 控制器。

控制器接受用户的输入并调用模型和视图去完成用户的需求。所以，当用户单击 Web 页面中的超链接和发送 HTML 表单时，控制器（如 Servlet）本身不输出任何东西和执行任何处理，它只是接收请求并决定调用哪个模型构件去处理请求，然后确定用哪个视图来显示模型处理返回的数据。

MVC 的处理过程是这样的：对于每一个用户输入的请求，首先被控制器接收，并决定由哪个模型来进行处理，然后模型通过业务逻辑层处理用户的请求并返回数据，最后控制器用相应的视图格式化模型返回的数据，并通过显示页面呈现给用户。

MVC 的这种特殊的设计结构，给应用开发带来了很多便捷，通过 MVC 架构的使用，大大地提高了 Web 应用的开发效率。具体来说，MVC 设计架构主要有以下几个方面的优点：

1) 低耦合性。由于视图层和业务层分离，使得修改视图层代码时不需要重新编译模型和控制器的代码，同样一个应用的业务流程或者业务规则的改变只需要改动 MVC 的模型层即可。因为模型与控制器和视图相分离，所以很容易改变应用程序的数据层和业务规则。

2) 高重用性和可适用性。由于技术的不断进步，现在访问应用程序可以有越来越多的方式。MVC 模式允许使用各种不同样式的视图来访问同一个服务器端的代码。它包括任何 Web

(HTTP) 浏览器或者无线浏览器 (WAP)。例如, 用户可以通过计算机也可通过手机来订购某样产品, 虽然订购的方式不一样, 但处理订购产品的方式是一样的。由于模型返回的数据没有进行格式化, 所以同样的构件能被不同的界面使用。例如, 很多数据可能用 HTML 来表示, 但是也有可能用 WAP 来表示, 而这些表示所需要的命令仅是改变视图层的实现方式, 而控制层和模型层无需做任何改变。

3) 较低的生命周期成本。MVC 使得开发和维护用户接口的技术含量降低。

4) 部署快速。使用 MVC 模式可以相当大地缩减开发时间, 它使程序员 (开发人员) 集中精力于业务逻辑, 而界面程序员 (HTML 和 JSP 开发人员) 集中精力于表现形式上。

5) 可维护性。分离视图层和业务逻辑层也使得 Web 应用更易于维护和修改。

6) 有利于软件工程化管理。由于采用了分层思想, 每一层不同的应用具有某些相同的特征, 有利于通过工程化、工具化管理程序代码。

9.2 网络设备

9.2.1 交换机与路由器有什么区别

交换机是一种基于 MAC (网卡的硬件地址) 识别, 能完成封装转发数据包功能的网络设备。它具有流量控制能力, 主要用于组建局域网。例如, 搭建一个公司网络, 一般会使用交换机。常见的交换机种类有以太网交换机、光纤交换机等。路由器是连接 Internet 中各局域网、广域网的网络设备。它是网络的枢纽, 是组成广域网的一个重要部分, 用于为数据包找到最合适的到达路径。

具体而言, 交换机与路由器的区别主要表现在以下 3 个方面:

(1) 工作层次不同。交换机一般工作在 OSI 模型的数据链路层, 而路由器工作在 OSI 模型的网络层。由于交换机工作在 OSI 模型的数据链路层, 所以它的工作原理比较简单, 而路由器工作在 OSI 模型的网络层, 可以得到更多的协议信息, 路由器可以做出更加智能的转发决策。

(2) 数据转发所依据的对象不同。交换机是利用物理地址来确定转发数据的目的地址, 而路由器则是利用 IP 地址来确定数据转发的地址。IP 地址是在软件中实现的, 描述的是设备所在的网络, 物理地址一般是指 MAC 地址, 它通常是硬件自带的, 由网卡生产商来分配的, 而且已经固化到了网卡中去, 一般来说是不可更改的 (可以通过工具来修改机器的 MAC 地址); 而 IP 地址则通常由网络管理员或系统自动分配。

(3) 传统的交换机只能分割冲突域, 不能分割广播域; 而路由器可以分割广播域。由交换机连接的网段仍属于同一个广播域, 广播数据包会在交换机连接的所有网段上传播, 在某些情况下会导致通信拥塞以及产生安全漏洞。连接到路由器上的网段会被分配成不同的广播域, 广播数据不会穿过路由器。虽然第三层以上交换机具有 VLAN 功能, 也可以分割广播域, 但是各子广播域之间是不能通信交流的, 它们之间的交流仍然需要路由器。

(4) 交换机负责同一网段的通信, 路由器负责不同网段的通信。路由器提供了防火墙的服务, 它仅仅转发特定地址的数据包, 不传送不支持路由协议的数据包, 也不传送未知目标网络数据包, 从而可以防止广播风暴。

引申: 集线器 (Hub) 与交换机的区别是什么?

集线器实质上是一个中继器, 它与网卡、双绞线等传输介质一样, 是数据通信系统中的设备, 它工作在 OSI 模型的物理层, 对接收到的信号进行放大, 同时把所有结点集中在以它为

中心的结点上。具体而言,集线器与交换机有以下4个方面的不同:

(1) 工作位置不同。集线器工作在 OSI 模型的物理层,而交换机工作在 OSI 模型的数据链路层。集线器只是对数据的传输起到同步、放大和整形的作用,对数据传输中的短帧、碎片等无法进行有效的处理,不能保证数据传输的完整性和正确性,它类似于一个大的总线型局域网;而交换机不但可以对数据的传输做到同步、放大和整形,而且可以过滤短帧、碎片,对封装数据包进行转发等。

(2) 工作方式不同。集线器是一种广播模式,当集线器的某个端口工作时,其他所有端口都能够收听到信息,容易产生广播风暴,并且每一个时刻只有一个端口发送数据,而且集线器的安全性不好,所有的网卡都能接收到它所发的数据,只是非目的地网卡丢弃了信包。当交换机工作的时候,只有发出请求的端口和目的端口之间相互响应而不影响其他端口,因此交换机能够隔离冲突域和有效地抑制广播风暴的产生。

(3) 带宽不同。不管有多少个端口,集线器的所有端口都是共享一条带宽,在同一时刻只能有两个端口传送数据,其他端口只能等待,同时集线器只能工作在半双工模式下;而交换机的每个端口都有一条独占的带宽,当两个端口工作时并不影响其他端口的工作,同时交换机不但可以工作在半双工模式下而且可以工作在全双工模式下。

(4) 性能不同。交换机以 MAC 地址进行寻址,有一定的额外寻址开销,在数据流量小时,时延相对数据传输时间而言可能较大;集线器以广播方式传输数据,流量小时性能下降不明显,适合于共享总线型结构局域网。

9.2.2 路由表的功能有哪些

路由表是指路由器或者其他互联网网络设备上存储的表,它决定如何将包从一个子网传递到另一个子网。该表中存有到达特定网络终端的路径,在某些情况下,还有一些与路径相关的度量。路由器的主要工作就是为经过路由器的每个数据帧寻找一条最佳传输路径,并将该数据有效地传送到目的站点。

路由表可以是由系统管理员固定设置好的,也可以由系统动态修改,可以由路由器自动调整,也可以由主机控制。

静态路由是指由网络管理员手工配置的路由信息。当网络的拓扑结构或链路的状态发生变化时,网络管理员需要手工去修改动态路由表中相关的静态路由信息。静态路由信息在默认情况下是私有的,不会传递给其他的路由器。当然,网管员也可以通过对路由器进行设置使之成为共享的。静态路由一般适用于比较简单的网络环境,在这样的环境中,网络管理员易于清楚地了解网络的拓扑结构,便于设置正确的路由信息。

由系统管理员事先设置好固定的路由表称为静态路由表,一般是在系统安装时就根据网络的配置情况预先设定的,它不会随着未来网络结构的改变而改变。

动态路由是指路由器能够自动地建立自己的动态路由表,并且能够根据实际情况的变化适时地进行调整。动态路由机制的运作依赖路由器的两个基本功能:对路由表的维护,路由器之间适时的路由信息交换。

动态路由表是路由器根据网络系统的运行情况而自动调整的路由表。路由器根据路由选择协议提供的功能,自动学习和记忆网络运行情况,在需要时自动计算数据传输的最佳路径。路由器通常依靠所建立及维护的动态路由表来决定如何转发。动态路由表能力是指路由表内所容纳路由表项数量的极限。由于 Internet 上执行 BGP 的路由器通常拥有数十万条路由表项,所以该项目也是路由器能力的重要体现。

9.3 网络协议

9.3.1 TCP 和 UDP 的区别有哪些

传输层协议主要有 TCP 与 UDP。UDP (User Datagram Protocol) 提供无连接的通信, 不能保证数据包被发送到目标地址, 典型的即时传输少量数据的应用程序通常使用 UDP。TCP (Transmission Control Protocol) 是一种面向连接 (连接导向) 的、可靠的、基于字节流的通信协议, 它为传输大量数据或为需要接收数据许可的应用程序提供连接定向和可靠的通信。

TCP 连接就像打电话, 用户拨特定的号码, 对方在线并拿起电话, 然后双方进行通话, 通话完毕之后再挂断, 整个过程是一个相互联系、缺一不可的过程。而 UDP 连接就像发短信, 用户短信发送给对方, 对方有没有收到信息, 发送者根本不知道, 而且对方是否回答也不知道, 对方对信息发送者发送消息也是一样。

TCP 与 UDP 都是一种常用的通信方式, 在特定的条件下发挥不同的作用。具体而言, TCP 和 UDP 的区别主要表现为以下几个方面:

(1) TCP 是面向连接的传输控制协议, 而 UDP 提供的是无连接的数据报服务。

(2) TCP 具有高可靠性, 确保传输数据的正确性, 不出现丢失或乱序; UDP 在传输数据前不建立连接, 不对数据报进行检查与修改, 无需等待对方的应答, 所以会出现分组丢失、重复、乱序, 应用程序需要负责传输可靠性方面的所有工作。

(3) TCP 对系统资源要求较多, UDP 对系统资源要求较少。

(4) UDP 具有较好的实时性, 工作效率较 TCP 高。

(5) UDP 的段结构比 TCP 的段结构简单, 因此网络开销也小。

UDP 比 TCP 的效率要高, 为什么 TCP 还能够保留呢? 其实, TCP 和 UDP 各有所长、各有所短, 适用于不同要求的通信环境, 有些环境采用 UDP 确实高效, 而有些环境需要可靠的连接, 此时采用 TCP 则更好。在提及 TCP 的时候, 一般也提及 IP。IP 协议是一种网络层协议, 它规定每个互联网上的计算机都有一个唯一的 IP 地址, 这样数据包就可以通过路由器的转发到达指定的计算机, 但 IP 并不保证数据传输的可靠性。

9.3.2 什么叫三次握手? 什么叫四次断开

TCP 是一个面向连接的协议, 所谓面向连接是指通信双方任何一方向对方发送数据前必须先建立安全通道, 就像打电话一样, 必须要等到对方的手机响铃, 并且对方接听电话时, 才能与对方通信。而 UDP 则不是面向连接的协议, 基于 UDP 的通信双方不需要事先与对方协商并建立连接, 也不管对方的 IP 地址与端口号是否存在, 就直接发送数据, 这个处理方式有点像手机发短信, 不管对方手机是否停机或者关机, 只管发送信息, 不管对方是否能收到消息。

在 TCP/IP 中, 采用三次握手来建立一次连接, 第一次握手: 建立连接时, 客户端发送 SYN 包 (假如序列号 SEQ=100) 给服务器, 并进入 SYN_SEND 状态, 等待服务器的确认。第二次握手: 服务器收到 SYN 包之后, 必须确认客户端, 所以就要发送 ACK 包 (ACK=101), 同时服务器还必须发送 SYN 包 (序列号 SEQ=300) 等客户端的确认, 此时服务器进入 SYN_RECV 状态。第三次握手: 客户端接收到 SYN+ACK 包之后, 向服务器发送确认包 (ACK=301), 该包发送完毕, 此时客户端与服务器进入 ESTABLISHED, 两者就可以进行数据交换了, 完成三次握手。图 9-4 所示为三次握手的图例。

由于 TCP 连接是全双工的，因此每个方向上都必须单独进行关闭，当一方完成数据发送任务后就能发送一个 FIN 来终止这个方向的连接。一个 TCP 连接在收到一个 FIN 之后仍然能够发送数据，首先进行关闭的一方将主动执行关闭，而另一方将会执行被动关闭。

(1) 客户端发送数据完毕之后，发送一个 FIN，提出断开连接要求。

(2) 服务器收到该 FIN 包后，对其作出响应，发送一个 ACK 包，确认这一方向的连接将关闭。

(3) 等服务器的应用程序做好关闭准备时，服务器反方向发送一个 FIN 包给客户端，请求关闭连接请求。

(4) 客户机对服务器发送的请求进行确认，并发送 ACK 包。

图 9-5 所示为四次断开的图例。

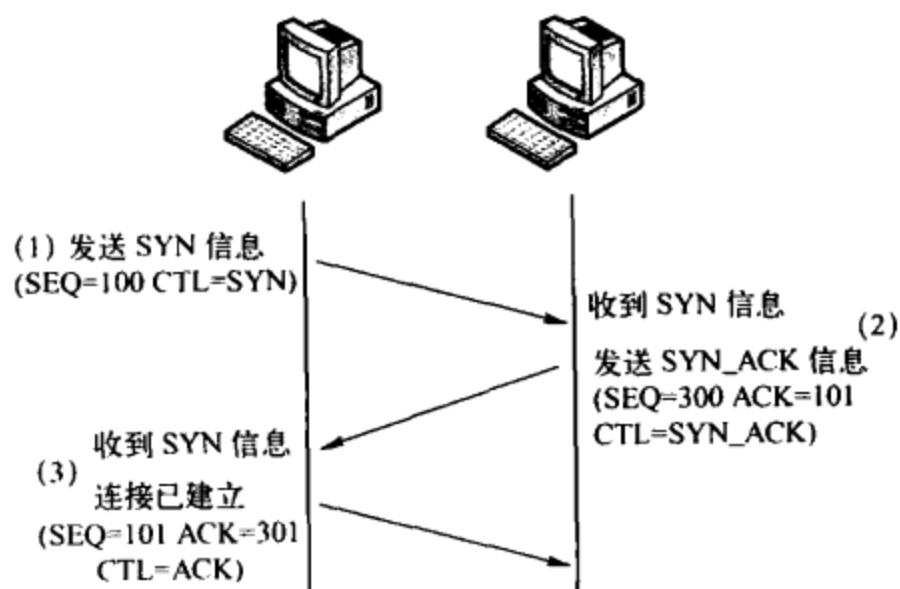


图 9-4 三次握手

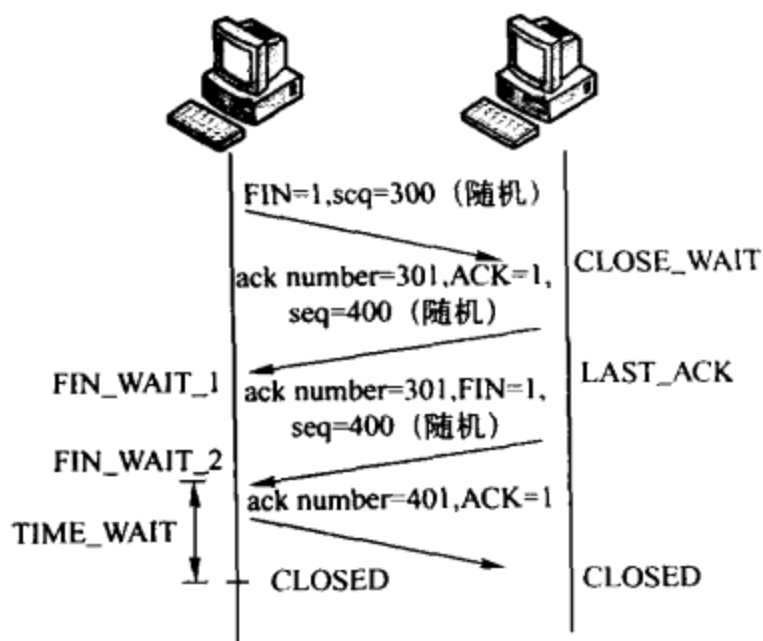


图 9-5 四次断开

常见的网络状态及其描述见表 9-1。

表 9-1 常见网络状态及其描述

| 状 态 | 描 述 |
|--------------------------|---|
| CLOSED | 表示初始状态 |
| LISTEN | 表示服务器端的某个 SOCKET 处于监听状态，可以接受连接 |
| SYN_RCVD | 表示接收到了 SYN 报文 |
| SYN_SENT | 表示客户端已发送 SYN 报文 |
| ESTABLISHED | 表示连接已经建立了 |
| FIN_WAIT_1 FIN_WAIT_2 | 表示等待对方的 FIN 报文。FIN_WAIT_1 状态实际上是当 SOCKET 在 ESTABLISHED 状态时，它想主动关闭连接，向对方发送了 FIN 报文，此时该 SOCKET 即进入到 FIN_WAIT_1 状态。而当对方回应 ACK 报文后，则进入到 FIN_WAIT_2 状态 |
| TIME_WAIT | 表示收到了对方的 FIN 报文，并发送出了 ACK 报文，就等 2MSL 后即可回到 CLOSED 可用状态了 |
| CLOSING | 表示双方都正在关闭 SOCKET 连接 |
| CLOSE_WAIT | 表示在等待关闭 |
| LAST_ACK | 被动关闭一方在发送 FIN 报文后，最后等待对方的 ACK 报文。当收到 ACK 报文后，就可以进入到 CLOSED 可用状态了 |

在第四步的时候，为什么需要 TIME_WAIT？HOST1 发送的 ACK 可能丢失并导致 HOST2 重新发送 FIN 消息，TIME_WAIT 维护连接状态。如果执行主动关闭的一方 HOST1 不进入到 TIME_WAIT 状态就关闭连接，当重传的 FIN 消息到达时，因为 TCP 已经不再有连接

的信息了，所以就用 RST（重新启动）消息应答，导致 HOST2 进入错误的状态而不是有序终止状态。如果发送最后 ACK 消息的一方处于 TIME_WAIT 状态并仍然记录着连接的信息，它就可以正确地响应对等方 HOST2 的 FIN 消息了。

其次，TIME_WAIT 为连接中“离群的段”提供从网络中消失的时间，通常情况下，因为 TCP 仅仅丢弃该数据并响应 RST 消息，所以这不会造成任何问题。当 RST 消息到达发出延时段的主机时，因为该主机也没有记录连接的任何信息，所以它也丢弃该段。然而，如果两个相同主机之间又建立了一个具有相同端口号的新连接，那么离群的段就可能被看成是新连接的；如果离群的段中数据的任何序列号恰恰在新连接的当前接收窗口中，数据就会被重新接收，其结果就是破坏新连接。

引申 1：TCP 为什么需要三次握手，采用两次握手可以吗？

建立连接的过程是利用客户服务器模式，假设主机 A 为客户端，主机 B 为服务器端。

(1) TCP 的三次握手过程：主机 A 向 B 发送连接请求，主机 B 对收到的主机 A 的报文段进行确认，主机 A 再次对主机 B 的确认进行确认。

(2) 采用三次握手是为了防止失效的连接请求报文段突然又传送到主机 B，因而产生错误。失效的连接请求报文段是指：主机 A 发出的连接请求没有收到主机 B 的确认，于是经过一段时间后，主机 A 又重新向主机 B 发送连接请求，且建立成功，顺序完成数据传输。考虑这样一种特殊情况，主机 A 第一次发送的连接请求并没有丢失，而是因为网络结点导致延迟达到主机 B，主机 B 以为是主机 A 又发起的新连接，于是主机 B 同意连接，并向主机 A 发回确认，但是此时主机 A 根本不会理会，主机 B 就一直在等待主机 A 发送数据，导致主机 B 的资源浪费。

(3) 采用两次握手不行，原因就是上面说的失效的连接请求的特殊情况。

引申 2：为什么建立连接协议是三次握手，而关闭连接却是四次握手？

因为服务端的 LISTEN 状态下的 SOCKET 收到 SYN 报文的建立连接请求后，它可以把 ACK 和 SYN（ACK 起应答作用，而 SYN 起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的 FIN 报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你未必会马上会关闭 SOCKET，也即你可能还需要发送一些数据给对方之后，再发送 FIN 报文给对方来表示同意现在可以关闭连接了，所以这里的 ACK 报文和 FIN 报文多数情况下都是分开发送的。

9.3.3 什么是 ARP/RARP

ARP（Address Resolution Protocol，地址解析协议）是一个位于 TCP/IP 协议栈中的低层协议，它用于映射计算机的物理地址与网络 IP 地址。在 Internet 分布式环境中，每个主机都被分配了一个 32 位的网络地址，此时就存在将计算机的 IP 地址与物理地址之间的转换问题。ARP 所要做的工作就是在主机发送帧前，根据目标 IP 地址获取 MAC 地址，以保证通信过程的顺畅。

其具体过程如下：首先，每台主机都会在自己的 ARP 缓冲区中建立一个 ARP 列表，用于存储 IP 地址与 MAC 地址的对应关系。然后当源主机需要将一个数据包发送到目标主机时，会先检查自己的 ARP 列表是否存在该 IP 地址对应的 MAC 地址。如果存在则直接将数据包发送到该 MAC 地址；如果不存在，就向本地网段发起一个 ARP 请求的广播包，用于查询目标主机对应的 MAC 地址。此 ARP 请求数据包里包括源主机的 IP 地址、硬件地址以及目标主机的 IP 地址等。网络中所有的主机收到这个 ARP 请求之后，会检查数据包中的目的 IP 是否与

自己的 IP 地址一致, 如果不同就忽略此数据包; 如果相同, 该主机将发送端的 MAC 地址与 IP 地址添加到自己的 ARP 列表中。如果 ARP 列表中已经存在该 IP 地址的相关信息, 则将其覆盖掉, 接着给源主机发送一个 ARP 响应包, 告诉对方自己是它所需要查找的 MAC 地址。最后源主机收到这个 ARP 响应包后, 将得到的目的主机的 IP 地址和 MAC 地址添加到自己的 ARP 列表中, 并利用此信息开始数据的传输。如果源主机一直没有收到 ARP 响应包, 则表示 ARP 查询失败。

RARP 与 ARP 工作方式相反。RARP 发出要反向解析的物理地址并希望返回其对应的 IP 地址, 应答包括由能够提供所需信息的 RARP 服务器发出的 IP 地址。RARP 获取 IP 地址的过程如下: 首先需要知道自己 IP 地址的机器向另一台机器上的服务器发送请求, 并等待服务器发出响应, 开始不知道服务器的物理地址, 所以通过广播。一旦通过广播对地址的请求, 就必须唯一标识自己的硬件标识 (如 CPU 序列号), 这个标识能让可执行程序容易获得。源主机收到从 RARP 服务器的响应消息后, 就可以利用得到的 IP 地址进行通信。

9.3.4 IP Phone 的原理是什么?都用了哪些协议

IP 电话 (又称 IP Phone) 是通过互联网或其他使用 IP 技术的网络来实现电话通信的。它是一种全新的通信技术, 建立在 IP 技术上的分组化、数字化传输技术基础之上。其原理是通过语音压缩算法对语音数据进行压缩编码处理, 然后把这些语音数据按 IP 等相关协议进行打包, 经过 IP 网络把数据包传输到接收方, 再把这些语音数据包串起来, 经过解码解压处理后, 恢复成原来的语音信号, 从而达到由 IP 网络传送语音的目的。VOIP (Voice Over Internet Protocol) 使用的协议有 H.323 协议簇、SIP、Skype 协议、H.248 和 MGCP。

9.3.5 Ping 命令是什么

Ping (Packet Internet Grope, 因特网包探索器) 是一个用于测试网络连接量的程序。它使用的是 ICMP, Ping 发送一个 ICMP (Internet Control and Message Protocol, 因特网控制报文协议) 请求消息给目的地并报告是否收到所希望的 ICMP 应答。

ICMP 是 TCP/IP 协议簇的一个子协议, 用于在 IP 主机、路由器之间传递控制消息。它是用来检查网络是否通畅或者网络连接速度的命令。

由于网络上的机器都有唯一确定的 IP 地址, 当给目标 IP 地址发送一个数据包 (包括对方的 IP 地址和自己的地址以及序列数) 时, 对方就要返回一个同样大小的数据包 (包括双方地址), 根据返回的数据包可以确定目标主机的存在, 可以初步判断目标主机的操作系统等。

例如, 当执行命令 `ping www.xidian.edu.cn`, 通常是通过 DNS 服务器, 如果这里出现故障, 则表示 DNS 服务器的 IP 地址配置不正确或 DNS 服务器有故障。也可以利用该命令实现域名对 IP 地址的转换功能。例如, `ping` 某一网络地址 `www.baidu.com`, 出现: “Reply from 119.75.217.109: bytes=32 time=31ms TTL=48” 则表示本地与该网络地址之间的线路是畅通的; 如果出现 “Request timed out”, 则表示此时发送的小数据包不能到达目的地, 此种情况可能由两种原因导致, 第一种是网络不通, 第二种是网络连通状况不佳。此时可以使用带参数的 Ping 来确定是哪一种情况。例如, `ping www.baidu.com -t -w 3000` 不断地向目的主机发送数据, 并且响应时间增大到 3000ms, 此时如果都是显示 “Request timed out”, 则表示网络之间确实不通; 如果不是全部显示 “Request timed out” 则表示此网站还是通的, 只是响应时间长或通信状况不佳。

由于 ping 使用的是 ICMP, 有些防火墙软件会屏蔽掉 ICMP, 所以有时候 ping 的结果只能

做为参考, ping 不通并不能就一定说明对方 IP 不存在。但一般而言, 在通过 ping 进行网络故障判断时, 如果 ping 运行正确, 大体上就可以排除网络访问层、网卡、Modem 的输入输出线路、电缆和路由器等存在的故障, 从而减小了问题的范围。

9.3.6 基本的 HTTP 流程有哪些

HTTP 是 Hyper Text Transfer Protocol (超文本传输协议) 的缩写, 其主要负责服务器与浏览器之间的通信。HTTP 把客户端浏览器的请求发送到服务器, 并把响应的网页内容由服务器返回到客户端浏览器。

一次完整的 HTTP 流程一般包括以下几个步骤:

- (1) 打开 HTTP 连接。因为 HTTP 是一种无状态协议, 所以每一个请求都需要建立一个新的连接。
- (2) 初始化方法请求。包含一些类型的方法指示符, 它们用来描述调用什么方法和需要什么参数。
- (3) 设置 HTTP 请求头。包含要传输的数据类型和数据长度。
- (4) 发送请求。即将二进制流写入服务器。
- (5) 读取请求。调用目标 servlet 程序, 并接受 HTTP 请求数据。如果该次请求为客户端第一次请求, 则需要创建一个新的服务器对象实例。
- (6) 调用方法。提供了服务器端调用对象的方法。
- (7) 初始化响应方法。如果调用的方法出现异常, 客户将会收到出错信息; 否则, 发送返回类型。
- (8) 设置 HTTP 响应头。响应头中设置待发送的数据类型与长度。
- (9) 发送响应。服务器端发送二进制数据流给客户端作为响应。
- (10) 关闭连接。当响应结束后, 与服务器必须断开连接, 以保证其他请求能够与服务器建立连接。

9.4 网络编程

9.4.1 如何使用 Socket 编程

Socket 在计算机中提供了一个通信端口, 可以通过这个端口与任何一个具有 Socket 接口的计算机通信, 应用程序在网络上传输、接收的信息都通过这个 Socket 接口来实现。

Socket 通信原理如下: 服务器端实现监听连接, 客户端实现发送连接请求, 建立连接后, 发送和接收数据进行通信。具体而言, 首先服务器端建立一个 Socket, 设置好本机的 IP 和监听的端口并与 Socket 进行绑定, 开始监听连接请求, 当接收到连接请求后, 发送确认, 同客户端建立连接, 开始与客户端进行通信。同时客户端建立一个 Socket, 设置好服务器端的 IP 和提供服务的端口, 发出连接请求, 接收到服务的确认后, 尽力连接, 开始与服务器进行通信。服务器端和客户端的连接及它们之间的数据传送均采用同步方式 (要链接一个 ws2_32.lib 的库文件, 头文件 winsock2.h, dll 文件 Ws2_32.dll)。采用 UDP 编程则不一样, UDP 编程的发送端只管发送就可以了, 不用检查网络的连接状态。

图 9-6 所示为 Socket 通信原理图。

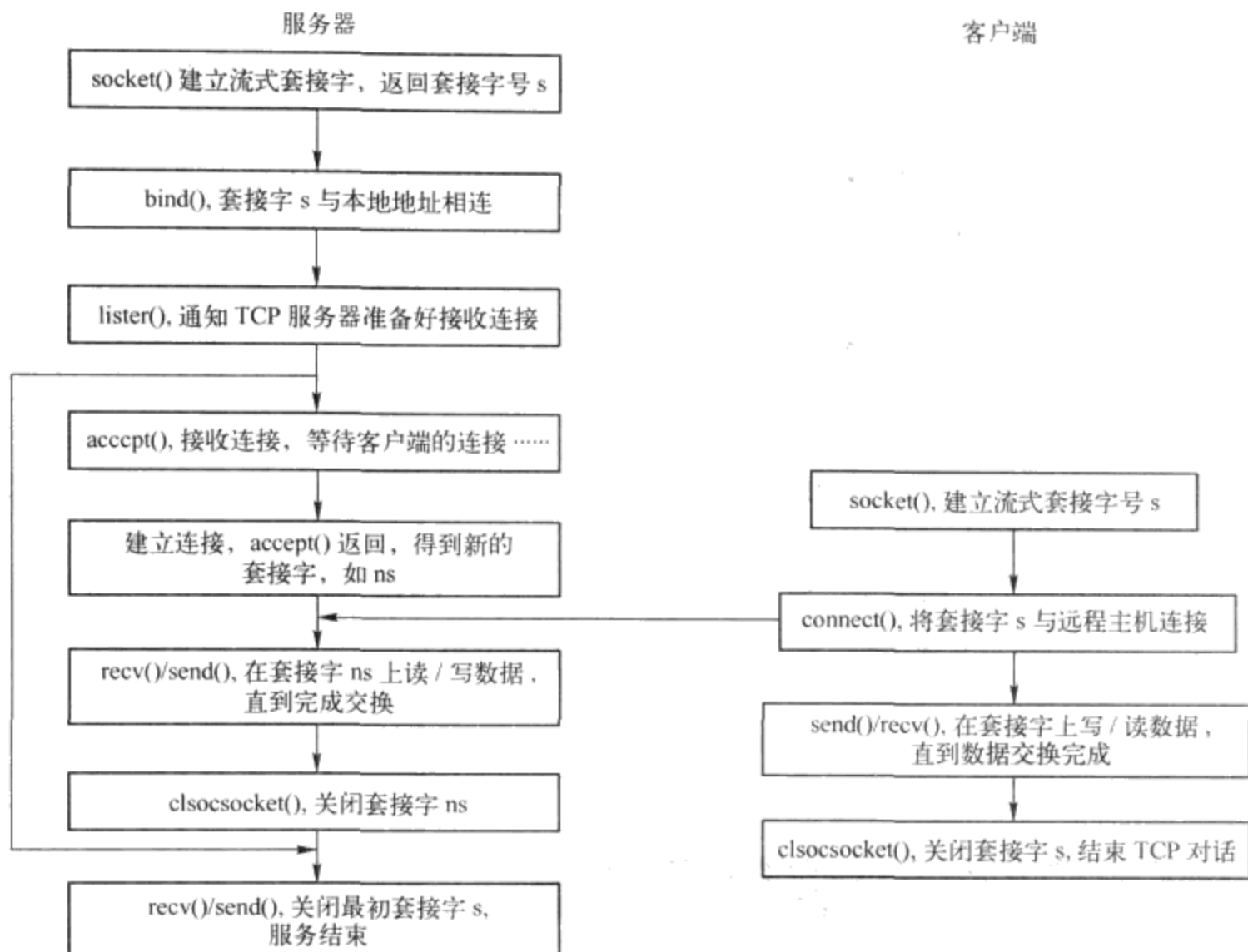


图 9-6 Socket 通信原理图

在服务器端有一个控制台程序（或者 Windows 服务）与多个客户端程序通信，其中主线程有一个 Socket 绑定在一个固定端口上，负责监听客户端的 Socket 信息。每当启动一个客户端程序，客户端发送来一个 Socket 连接请求，服务器端就新开启一个线程，并在其中创建一个 Socket 与该客户端的 Socket 通信，直到客户端程序关闭，结束该线程，主线程中的 Socket 在应用程序退出时关闭。

9.4.2 阻塞模式和非阻塞模式有什么区别

使用 Socket 编程实现数据传输的过程，通常的默认设置假设套接字是阻塞的。每一个 TCP 套接字有一个发送缓冲区，当应用进程调用 write 操作时，内核从应用进程的缓冲区中复制数据到套接字的发送缓冲区。如果套接字的发送缓冲区无法完全容纳应用程序的所有数据，即应用进程的缓冲区大于套接字发送缓冲区或套接字发送缓冲区还有其他数据，应用进程将会被挂起，内核将不从 write 系统调用返回，直到应用进程缓冲区中的所有数据都复制到套接字发送缓冲区。因此，从写一个 TCP 套接字的 write 调用成功返回仅仅表示可以重新使用应用进程的缓冲区，它并不表示对端的应用进程已经接收到了数据。

下面从发送和接收两方面说明阻塞模式和非阻塞模式的区别。

(1) 发送操作：write、writev、send、sendto、sendmsg。

对于一个 TCP 套接字，内核将从应用进程的缓冲区向该套接字的发送缓冲区复制数据。对于阻塞的套接字，如果其发送缓冲区中没有空间，进程将挂起，直到有空间为止。

对于一个非阻塞的 TCP 套接字，如果其发送缓冲区中根本没有空间，发送函数调用将立即返回一个 EWOULDBLOCK 错误。如果其发送缓冲区中有一些空间，返回值将是内核能够

复制到该缓冲区中的字节数。

UDP 套接字不存在真正的发送缓冲区。内核只是复制应用进程数据并把它沿协议栈向下传送，依次加上 UDP 头部和 IP 头部。因此，对一个阻塞的 UDP 套接字，发送函数调用将不会因为与 TCP 套接字一样的原因而阻塞，不过有可能会因为其他的原因而阻塞。

(2) 接收操作：`read`、`readv`、`recv`、`recvfrom`、`recvmsg`。

如果某个进程对一个阻塞的 TCP 套接字调用这些输入函数之一，而且该套接字的接收缓冲区中没有数据可读，该进程将被挂起，直到到达一些数据。TCP 是字节流协议，只要到达一些数据，该进程就会被唤醒：这些数据既可能是单个字节，也可以是一个完整的 TCP 分节中的数据。

UDP 是数据报协议，如果一个阻塞的 UDP 套接字的接收缓冲区为空，对它调用接收函数的进程将被挂起，直到到达一个 UDP 数据报。

对于非阻塞的套接字，如果接收操作不能被满足（对于 UDP 套接字即有一个完整的数据报可读），相应的调用将立即返回一个 `EWOULDBLOCK` 错误。

9.5 网络其他问题

9.5.1 常用的网络安全防护措施有哪些

计算机网络由于分布式特性，使得它容易受到来自网络的攻击。网络安全是指“在一个网络环境里，为数据处理系统建立和采取的技术与管理的安全保护，利用网络管理控制和技术措施保护计算机软件、硬件数据不因为偶然或恶意的原因而遭到破坏、更改和泄露”。常见的网络安全防护措施有加密技术、验证码技术、认证技术、访问控制技术、防火墙技术、网络隔离技术、入侵检测技术、防病毒技术、数据备份与恢复技术、VPN 技术、安全脆弱性扫描技术、网络数据存储、备份及容灾规划等。

(1) 加密技术。数据在传输过程中有可能因攻击者或入侵者的窃听而失去保密性。加密技术是最常用的保密安全手段之一，它对需要进行伪装的机密信息进行变换，得到另外一种看起来似乎与原有信息不相关的表示。合法用户可以从这些信息中还原出原来的机密信息，而非法用户如果试图从这些伪装后的信息中分析出原有的机密信息，要么这种分析过程根本是不可能的，要么代价过于巨大，以至于无法进行。

(2) 验证码技术。普遍的客户端交互，如留言本、会员注册等仅是按照要求输入内容，但网络上有很多非法应用软件，如注册机，可以通过浏览 Internet，扫描表单，然后在系统上频繁注册，频繁发送不良信息，造成不良的影响，或者通过软件不断地尝试，盗取用户密码。而通过使用验证码技术，使客户端输入的信息都必须经过验证，从而可以有效解决别有用心用户利用机器人（或恶意软件）自动注册、自动登录、恶意增加数据库访问、用特定程序暴力破解密码等问题。

所谓验证码是指将一串随机产生的数字或符号生成一幅图片，图片里加上一些干扰像素，由用户肉眼极易识别其中的验证码信息，输入表单提交网络应用程序验证，验证成功后才能使用某项功能。放在会员注册、留言本等所有客户端提交信息的页面，要提交信息，必须要输入正确的验证码，从而可以防止不法用户用软件频繁注册、频繁发送不良信息等。

使用验证码技术必须保证所有客户端交互部分都输入验证码，测试提交信息时不输入验证

码, 或者故意输入错误的验证码, 如果信息都不能交, 说明验证码有效, 同时在验证码输入正确下提交信息, 如果能提交, 说明验证码功能已完善。

(3) 认证技术。认证技术是信息安全的一项重要内容, 很多情况下, 用户并不要求信息保密, 只要确认网络服务器或在线用户不是假冒的, 自己与他们交换的信息未被第三方修改或伪造, 且网上通信是安全的。

认证是指核实真实身份的过程, 是防止主动攻击的重要技术之一, 是一种可靠地证实被认证对象(包括人和事)是否名副其实或者是否有效的过程, 因此也称为鉴别或验证。认证技术的作用主要是通过一定的手段在网络上弄清楚对象是谁, 具有什么样的特征(特征具有唯一性)。认证可以是某个个人、某个机构代理、某个软件(如股票交易系统), 这样可以确定对象的真实性, 防止假冒、篡改等行为。

(4) 访问控制技术。网络中拥有各种资源, 通常可以是被调用的程序、进程, 要存取的数据、信息, 要访问的文件、系统, 或者是各种各样的网络设备, 如打印机、硬盘等。网络中的用户必须根据自己的权限范围来访问网络资源, 从而保证网络资源受控地、合法地使用。

访问控制是在身份认证的基础上针对越权使用资源的防范(控制)措施, 是网络安全防范和保护的主要策略。其主要任务是防止网络资源被非法使用、非法访问和不慎操作所造成破坏。它也是维护网络系统安全、保护网络资源的重要手段。

实现访问控制的关键是采用何种访问控制策略。目前主要有 3 种不同类型的访问控制策略: 自主访问控制(DAC)、强制访问控制(MAC)和基于角色的访问控制(RBAC)。目前 DAC 应用最多, 主要采用访问控制表(ACL)实现, 如 Apache Web 服务器、JDK 开发平台都支持 ACL。

此外, 在路由器的许多其他配置任务中都需要使用访问控制列表, 如网络地址转换、按需拨号路由、路由重分布、策略路由等很多场合都需要访问控制列表。访问控制列表从概念上来讲并不复杂, 复杂的是对它的配置和使用, 许多初学者往往在使用访问控制列表时出现错误。

除了上述提及的网络安全技术外, 其他常见的安全技术还有防火墙技术、网络隔离技术、入侵检测技术、防病毒技术、数据备份与恢复技术、VPN(Virtual Private Network, 虚拟专用网络)技术、安全脆弱性扫描技术、物理安全技术、虚拟网络技术、漏洞扫描技术、主机防护技术、安全评估技术、安全审计技术、加强行政管理、完善规章制度、严格人员选任和法律介入等。但是没有一种安全技术可以完美解决网络上的所有安全问题, 各种安全技术必须相互关联, 相互补充, 形成网络安全的立体纵深、多层次防御体系。

9.5.2 什么是 SQL 注入式攻击

所谓 SQL 注入式攻击就是攻击者把 SQL 命令插入到 Web 表单的域或页面请求的查询字符串中, 欺骗服务器执行恶意的 SQL 命令。在某些表单中, 用户输入的内容直接用来构造动态 SQL 命令, 或作为存储过程的输入参数, 这类表单特别容易受到 SQL 注入式攻击。

例如, 对于一个站点 <http://www.thesite.com/News/details.jsp?id=2> 的页面, id 是查询参数, 通过 id 获取显示某条信息。在 JSP 程序中, 用 SQL 语句来读取该条新闻: “select * from news where id = ” + id, 正常执行的话, 只需要将 id 替换为参数 2 即可, 但是当非法用户将 id 的参数变为 id=2;drop database news 时, 则执行的 SQL 语句除了读取对应的新闻信息外, 还会执行 drop database news 信息, 可是后面这条语句是非法的。

由于 SQL 注入攻击利用的是合法的 SQL 语句, 使得这种攻击不能被防火墙检查出来, 而且由于对任何基于 SQL 语言标准的数据库都适用, 所以危害特别大。尽管如此, 目前防止

SQL 注入攻击的方法也非常多，具体而言，有以下一些方法：

(1) 在利用表单输入的内容构造 SQL 命令之前，对用户输入进行验证（利用正则表达式等）与替换。例如，替换单引号，即把所有单独出现的单引号改成两个单引号，防止攻击者修改 SQL 命令的含义。

(2) 避免使用解释程序，攻击者一般借以执行非法命令。

(3) 对查询字符串、用户登录名称、密码等进行加密处理。

(4) 删除用户输入内容中的所有连字符，防止攻击者顺利获得访问权限。

(5) 对于用来执行查询的数据库账户，限制其权限。

(6) 用存储过程来执行所有查询。

(7) 检查用户输入的合法性，确信输入的内容只包含合法的数据。数据检查应当在客户端和服务端都执行。

(8) 检查提取数据的查询所返回的记录数量。

9.5.3 电路交换技术、报文交换技术和分组交换技术有什么区别

网络交换技术共经历了 4 个发展阶段，电路交换技术、报文交换技术、分组交换技术和 ATM 技术。

公众电话网（PSTN 网）和移动网（包括 GSM 网和 CDMA 网）采用的都是电路交换技术。电路交换技术的基本特点是采用面向连接的方式，在双方进行通信之前，需要为通信双方分配一条具有固定带宽的通信电路，通信双方在通信过程中将一直占用所分配的资源，直到通信结束，并且在电路的建立和释放过程中都需要利用相关的信令协议。这种方式的优点是在通信过程中可以保证为用户提供足够的带宽，并且实时性强、时延小、交换设备成本较低；但同时带来的缺点是网络的带宽利用率不高，一旦电路被建立不管通信双方是否处于通话状态，分配的电路都一直被占用。

报文交换技术和分组交换技术类似，也是采用存储转发机制，但报文交换是以报文作为传送单元，由于报文长度差异很大，长报文可能导致很大的时延，并且对每个结点来说缓冲区的分配也比较困难，为了满足各种长度报文的需要并且达到高效的目的，结点需要分配不同大小的缓冲区，否则就有可能造成数据传送的失败。在实际应用中报文交换主要用于传输报文较短、实时性要求较低的通信业务，如公用电报网。报文交换比分组交换出现的要早一些，分组交换是在报文交换的基础上，将报文分割成分组进行传输，在传输时延和传输效率上进行了平衡，从而得到广泛的应用。

电路交换技术主要适用于传送语音相关的业务，这种网络交换方式对于数据业务而言，有着很大的局限性。1) 数据通信具有很强的突发性，峰值比特率和平均比特率相差较大，如果采用电路交换技术，若按峰值比特率分配电路带宽则会造成资源的极大浪费；如果按照平均比特率分配带宽，则会造成数据的大量丢失。2) 和语音业务比较起来，数据业务对时延没有严格的要求，但需要进行无差错的传输，而语音信号可以有一定程度的失真但实时性一定要高。分组交换技术就是针对数据通信业务的特点而提出的一种交换方式，它的基本特点是面向无连接而采用存储转发的方式，将需要传送的数据按照一定的长度分割成许多小段数据，并在数据之前增加相应的用于对数据进行选路和校验等功能的头部字段，作为数据传送的基本单元即分组。采用分组交换技术，在通信之前不需要建立连接，每个结点首先将前一结点送来的分组收下并保存在缓冲区中，然后根据分组头部中的地址信息选择适当的链路将其发送至下一个结点，这样在通信过程中可以根据用户的要求和网络的能力来动态分配带宽。分组交换比电路交

换的电路利用率高，但时延较大。

电路交换技术的优点是：数据传输可靠、迅速，且能够保持原有序列。电路交换技术的缺点是：一旦通信双方占有一条通道后，即使不传送数据，其他用户也不能使用，造成资源浪费。

报文交换技术的优点是不需要事先建立物理线路，它将发送的数据作为一个整体发给中间的交换设备。中间交换设备先将数据存储起来，然后选择一条合适的空闲线路将数据发给下一个交换设备，如此循环直到数据发送到目的结点。

报文交换技术的缺点是报文在经过结点时会产生延迟。分组交换技术的优点是减少了延迟，提高了网络的速度。还提供一定程度的差错检测和代码转换，而其缺点则是如果遇到拥塞比较严重的情况，等待转发可能导致很长的时延，甚至还会造成数据分组丢失。

9.5.4 相比 IPv4，IPv6 有什么优点

IP 地址是 Internet 上主机或路由器的数字标识，用来唯一地标识该设备。IPv4 (Internet Protocol version 4, 互联网协议版本 4) 是一个被广泛使用的互联网协议，而 IPv6 是下一版本的互联网协议。随着互联网的迅速发展，IPv4 定义的有限地址空间将被耗尽，地址空间的不足必将妨碍互联网的进一步发展。为了扩大地址空间，拟通过 IPv6 重新定义地址空间。

IPv6 采用 128 位地址长度，几乎可以不受限制地提供地址。IPv6 不仅解决了地址短缺的问题，它还考虑了在 IPv4 中存在的端到端 IP 连接、服务质量、安全性、多播、移动性、即插即用等。

相比 IPv4，IPv6 主要有以下几个方面的优点：

(1) 更大的地址空间。IPv4 中规定 IP 地址长度为 32，即有 $2^{32}-1$ 个地址；而 IPv6 中 IP 地址的长度为 128，即有 $2^{128}-1$ 个地址。

(2) 更小的路由表。IPv6 的地址分配遵循聚类原则，这使得路由器能在路由表中用一条记录表示一片子网，大大减小了路由器中路由表的长度，提高了路由器转发数据包的速度。

(3) 增强的组播支持以及对流的支持。这使得网络上的多媒体应用有了长足发展的机会，为服务质量控制提供了良好的网络平台。加入了对自动配置的支持。这是对 DHCP 的改进和扩展，使得网络（尤其是局域网）的管理更加方便和快捷。

(4) 更高的安全性。在使用 IPv6 的网络中用户可以对网络层的数据进行加密并对 IP 报文进行校验，这极大地增强了网络安全。

鉴于 IPv6 的诸多优点，经过一个较长的 IPv4 与 IPv6 共存的时期，IPv6 最终会完全取代 IPv4 在互联网上占据统治地位。

对于计算机系统而言，操作系统充当着基石的作用，它是连接计算机底层硬件与上层应用软件的桥梁，控制其他程序的运行，并且管理系统相关资源，同时提供配套的系统软件支持。对于专业的程序员而言，掌握一定的操作系统知识必不可少，因为不管面对的是底层嵌入式开发，还是上层的云计算开发，都需要使用到一定的操作系统相关知识。所以，对操作系统相关知识的考查是程序员面试笔试必考项之一。

10.1 进程管理

10.1.1 进程与线程有什么区别

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，它是系统进行资源分配和调度的一个独立单位。例如，用户运行自己的程序，系统就创建一个进程，并为它分配资源，包括各种表格、内存空间、磁盘空间、I/O 设备等，然后该进程被放入到进程的就绪队列，进程调度程序选中它，为它分配 CPU 及其他相关资源，该进程就被运行起来。

线程是进程的一个实体，是 CPU 调度和分配的基本单位，线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器、一组寄存器和栈），但是它可以与同属一个进程的其他的线程共享进程所拥有的全部资源。

在没有实现线程的操作系统中，进程既是资源分配的基本单位，又是调度的基本单位，它是系统中并发执行的单元。而在实现了线程的操作系统中，进程是资源分配的基本单位，但线程是调度的基本单位，是系统中并发执行的单元。

引入线程主要有以下 4 个方面的优点：

- (1) 易于调度。
- (2) 提高并发性。通过线程可以方便有效地实现并发。
- (3) 开销小。创建线程比创建进程要快，所需要的开销也更少。
- (4) 有利于发挥多处理器的功能。通过创建多线程，每个线程都在一个处理器上运行，从而实现应用程序的并行，使每个处理器都得到充分运行。

需要注意的是，尽管线程与进程很相似，但两者也存在着很大的不同，区别如下：

- (1) 一个线程必定属于也只能属于一个进程；而一个进程可以拥有多个线程并且至少拥有一个线程。
- (2) 属于一个进程的所有线程共享该线程的所有资源，包括打开的文件、创建的 Socket 等。不同的进程互相独立。
- (3) 线程又被称为轻量级进程。进程有进程控制块，线程也有线程控制块。但线程控制块比进程控制块小得多。线程间切换代价小，进程间切换代价大。
- (4) 进程是程序的一次执行，线程可以理解为程序中一段程序片段的执行。
- (5) 每个进程都有独立的内存空间，而线程共享其所属进程的内存空间。

引申：程序、进程与线程的区别是什么？

程序、进程与线程的区别见表 10-1。

表 10-1 程序、进程与线程对比

| 名 称 | 描 述 |
|-----|---|
| 程序 | 一组指令的有序结合 |
| 进程 | 具有一定独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的一个独立单元 |
| 线程 | 进程的一个实体，是 CPU 调度和分派的基本单元，是比进程更小的能独立运行的基本单元。本身基本上不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈）。一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行 |

简而言之，一个程序至少有一个进程，一个进程至少有一个线程。

10.1.2 线程同步有哪些机制

现在流行的进程线程同步互斥的控制机制，其实是由最原始、最基本的 4 种方法实现的：临界区、互斥量、信号量和事件。

（1）临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。

（2）互斥量：为协调共同对一个共享资源的单独访问而设计。只有拥有互斥对象的线程才有权限去访问系统的公共资源，因为互斥对象只有一个，所以能够保证资源不会同时被多个线程访问。互斥不仅能实现同一应用程序的公共资源安全共享，还能实现不同应用程序的公共资源安全共享。

（3）信号量：为控制一个具有有限数量的用户资源而设计。它允许多个线程在同一个时刻去访问同一个资源，但一般需要限制同一时刻访问此资源的最大线程数目。

（4）事件：用来通知线程有一些事件已发生，从而启动后继任务的开始。

10.1.3 内核线程和用户线程的区别

根据操作系统内核是否对线程可感知，可以把线程分为内核线程和用户线程。

内核线程建立和销毁都是由操作系统负责、通过系统调用完成的，操作系统在调度时，参考各进程内的线程运行情况做出调度决定。如果一个进程中没有就绪态的线程，那么这个进程也不会被调度占用 CPU。

和内核线程相对应的是用户线程，用户线程指不需要内核支持而在用户程序中实现的线程，其不依赖于操作系统核心，用户进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。用户线程多见于一些历史悠久的操作系统，如 UNIX 操作系统，不需要用户态/核心态切换，速度快，操作系统内核不知道多线程的存在，因此一个线程阻塞将使得整个进程（包括它的所有线程）阻塞。由于这里的处理器时间片分配是以进程为基本单位的，所以每个线程执行的时间相对减少。为了在操作系统中加入线程支持，采用了在用户空间增加运行库来实现线程，这些运行库被称为“线程包”，用户线程是不能被操作系统所感知的。

引入用户线程有以下 4 个方面的优势：

- （1）可以在不支持线程的操作系统中实现。
- （2）创建和销毁线程、线程切换等线程管理的代价比内核线程少得多。
- （3）允许每个进程定制自己的调度算法，线程管理比较灵活。
- （4）线程能够利用的表空间和堆栈空间比内核级线程多。

用户线程的缺点主要有以下两点:

(1) 同一进程中只能同时有一个线程在运行, 如果有一个线程使用了系统调用而阻塞, 那么整个进程都会被挂起。

(2) 页面失效也会产生类似的问题。

内核线程的优缺点刚好跟用户线程相反。实际上, 操作系统可以使用混合的方式来实现线程。

10.2 内存管理

10.2.1 内存管理有哪几种方式

常见的内存管理方式有块式管理、页式管理、段式和段页式管理。最常用的是段页式管理。

(1) 块式管理: 把主存分为一大块一大块的, 当所需的程序片断不在主存时就分配一块主存空间, 把程序片断 load 入主存, 就算所需的程序片段只有几个字节也只能把这一块分配给它。这样会造成很大的浪费, 平均浪费了 50% 的内存空间, 但是易于管理。

(2) 页式管理: 把主存分为一页一页的, 每一页的空间要比一块一块的空间小很多, 显然这种方法的空间利用率要比块式管理高很多。

(3) 段式管理: 把主存分为一段一段的, 每一段的空间又要比一页一页的空间小很多, 这种方法在空间利用率上又比页式管理高很多, 但是也有另外一个缺点。一个程序片断可能会被分为几十段, 这样很多时间就会被浪费在计算每一段的物理地址上。

(4) 段页式管理: 结合了段式管理和页式管理的优点。把主存先分成若干段, 每个段又分成若干页。段页式管理每取一数据, 要访问 3 次内存。

10.2.2 分段和分页的区别是什么

页是信息的物理单位, 分页是为了实现离散分配方式, 以消减内存的外零头, 提高内存的利用率; 或者说, 分页仅仅是由于系统管理的需要, 而不是用户的需要。

段是信息的逻辑单位, 它含有一组其意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。页的大小固定且由系统确定, 把逻辑地址划分为页号和页内地址两部分, 是由机器硬件实现的, 因而一个系统只能有一种大小的页面。段的长度却不固定, 决定于用户所编写的程序, 通常由编辑程序在对源程序进行编辑时, 根据信息的性质来划分。

分页的作业地址空间是一维的, 即单一的线性空间, 程序员只需利用一个记忆符, 即可表示一地址。分段的作业地址空间是二维的, 程序员在标识一个地址时, 既需给出段名, 又需给出段内地址。

10.2.3 什么是虚拟内存

虚拟内存简称虚存, 是计算机系统内存管理的一种技术。它是相对于物理内存而言的, 可以理解为“假的”内存。它使得应用程序认为它拥有连续可用的内存(一个连续完整的地址空间), 允许程序员编写并运行比实际系统拥有的内存大得多的程序, 这使得许多大型软件项目能够在具有有限内存资源的系统上实现。而实际上, 它通常被分割成多个物理内存碎片, 还有部分暂时存储在外部磁盘存储器上, 在需要进行数据交换。虚存比实存有以下好处:

(1) 扩大地址空间。无论段式虚存, 还是页式虚存, 或是段页式虚存, 寻址空间都比实存大。

(2) 内存保护。每个进程运行在各自的虚拟内存地址空间，互相不能干扰对方。另外，虚存还对特定的内存地址提供写保护，可以防止代码或数据被恶意篡改。

(3) 公平分配内存。采用了虚存之后，每个进程都相当于有同样大小的虚存空间。

(4) 当进程需要通信时，可采用虚存共享的方式实现。

不过，使用虚存也是有代价的，主要表现在以下几个方面：

(1) 虚存的管理需要建立很多数据结构，这些数据结构要占用额外的内存。

(2) 虚拟地址到物理地址的转换，增加了指令的执行时间。

(3) 页面的换入换出需要磁盘 I/O，这是很耗时间的。

(4) 如果一页中只有一部分数据，会浪费内存。

10.2.4 什么是内存碎片？什么是内碎片？什么是外碎片

内存碎片是由于多次进行内存分配造成的，当进行内存分配时，内存格式一般为：（用户使用段）（空白段）（用户使用段），当空白段很小的时候可能不能提供给用户足够需要的空间，可能夹在中间的空白段的大小为 5，而用户需要的内存大小 6，这样会产生很多的间隙造成使用效率的下降，这些很小的空隙叫碎片。

内碎片：分配给程序的存储空间没有用完，有一部分是程序不使用，但其他程序也没法用的空间。内碎片是处于区域内部或页面内部的存储块，占有这些区域或页面的进程并不使用这个存储块，而在进程占有这块存储块时，系统无法利用它，直到进程释放它，或进程结束时，系统才有可能利用这个存储块。

由于空间太小，小到无法给任何程序分配（不属于任何进程）的存储空间是外碎片。外部碎片是出于任何已分配区域或页面外部的空闲存储块，这些存储块的总和可以满足当前申请的长度要求，但是由于它们的地址不连续或其他原因，使得系统无法满足当前申请。

内碎片和外碎片是一对矛盾体，一种特定的内存分配算法，很难同时解决好内碎片和外碎片的问题，只能根据应用特点进行取舍。

10.2.5 虚拟地址、逻辑地址、线性地址、物理地址有什么区别

虚拟地址是指由程序产生的由段选择符和段内偏移地址组成的地址。这两部分组成的地址并没有直接访问物理内存，而是要通过分段地址的变换处理后才对应到相应的物理内存地址。

逻辑地址指由程序产生的段内偏移地址。有时直接把逻辑地址当成虚拟地址，两者并没有明确的界限。

线性地址是指虚拟地址到物理地址变换之间的中间层，是处理器可寻址的内存空间（称为线性地址空间）中的地址。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段基址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经过变换产生物理地址。若是没有采用分页机制，那么线性地址就是物理地址。

物理地址是指现在 CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果。

虚拟地址到物理地址的转化方法是与体系结构相关的，一般有分段与分页两种方式。以 x86 CPU 为例，分段分页都是支持的。内存管理单元负责从虚拟地址到物理地址的转化。逻辑地址是段标识+段内偏移量的形式，MMU 通过查询段表，可以把逻辑地址转化为线性地址。如果 CPU 没有开启分页功能，那么线性地址就是物理地址；如果 CPU 开启了分页功能，MMU 还需要查询页表来将线性地址转化为物理地址：逻辑地址（段表）→线性地址（页表）

→物理地址。

映射是一种多对一的关系，即不同的逻辑地址可以映射到同一个线性地址上；不同的线性地址也可以映射到同一个物理地址上。而且，同一个线性地址在发生换页以后，也可能被重新装载到另外一个物理地址上，所以这种多对一的映射关系也会随时间发生变化。

10.2.6 Cache 替换算法有哪些

数据可以存放在 CPU 或者内存中。CPU 处理快，但是容量少；内存容量大，但是转交给 CPU 处理的速度慢。为此，需要 Cache（缓存）来做一个折中。最有可能的数据先从内存调入 Cache，CPU 再从 Cache 读取数据，这样会快许多。然而，Cache 中所存放的数据不是 100%有用的。CPU 从 Cache 中读取到有用数据称为“命中”。

Cache 替换算法有随机算法、FIFO 算法、LRU 算法、LFU 算法和 OPT 算法。

(1) 随机算法 (RAND)。随机算法就是用随机数发生器产生一个要替换的块号，将该块替换出去，此算法简单、易于实现，而且它不考虑 Cache 块过去、现在及将来的使用情况。但是由于没有利用上层存储器使用的“历史信息”、没有根据访存的局部性原理，故不能提高 Cache 的命中率，命中率较低。

(2) 先进先出 (FIFO) 算法。先进先出 (First In First Out, FIFO) 算法是将最先进入 Cache 的信息块替换出去。FIFO 算法按调入 Cache 的先后决定淘汰的顺序，选择最早调入 Cache 的字块进行替换，它不需要记录各字块的使用情况，比较容易实现，系统开销小，其缺点是可能会把一些需要经常使用的程序块（如循环程序）也作为最早进入 Cache 的块替换掉，而且没有根据访存的局部性原理，故不能提高 Cache 的命中率。因为最早调入的信息可能以后还要用到，或者经常要用到，如循环程序。此法简单、方便，利用了主存的“历史信息”，但并不能说最先进入的就不经常使用，其缺点是不能正确反映程序局部性原理，命中率不高，可能出现一种异常现象。例如，Solar-16/65 机 Cache 采用组相连方式，每组 4 块，每块都设定一个两位的计数器，当某块被装入或被替换时该块的计数器清为 0，而同组的其他各块的计数器均加 1，当需要替换时就选择计数值最大的块被替换掉。

(3) 近期最少使用 (LRU) 算法。近期最少使用 (Least Recently Used, LRU) 算法是将近期最少使用的 Cache 中的信息块替换出去。该算法较先进先出算法要好一些，但此法也不能保证过去不常用将来也不常用。

LRU 算法是依据各块使用的情况，总是选择那个最近最少使用的块被替换。这种方法虽然比较好地反映了程序局部性规律，但是这种替换方法需要随时记录 Cache 中各块的使用情况，以便确定哪个块是近期最少使用的块。LRU 算法相对合理，但实现起来比较复杂，系统开销较大。通常需要对每一块设置一个称为计数器的硬件或软件模块，用以记录其被使用的情况。

实现 LRU 策略的方法有多种。下面简单介绍计数器法、寄存器栈法及硬件逻辑比较对法的设计思路。

计数器方法：缓存的每一块都设置一个计数器。计数器的操作规则如下：

1) 被调入或者被替换的块，其计数器清“0”，而其他的计数器则加“1”。

2) 当访问命中时，所有块的计数值与命中块的计数值要进行比较，如果计数值小于命中块的计数值，则该块的计数值加“1”；如果块的计数值大于命中块的计数值，则数值不变。最后将命中块的计数器清为“0”。

3) 需要替换时，则选择计数值最大的块被替换。

(4) 最优替换算法 (OPT 算法)。使用最优化替换算法 (OPTimal replacement algorithm) 时必须先执行一次程序，统计 Cache 的替换情况。有了这样的先验信息，在第二次执行该程序时便可以用最有效的方式来替换，以达到最优的目的。

前面介绍的几种页面替换算法主要是以主存储器中页面调度情况的历史信息为依据的，它假设将来主存储器中的页面调度情况与过去一段时间内主存储器中的页面调度情况是相同的，显然，这种假设不总是正确的。最好的算法应该是选择将来最久不被访问的页面作为被替换的页面，这种替换算法的命中率一定是最高的，它就是最优替换算法。

要实现 OPT 算法，唯一的办法是让程序先执行一遍，记录下实际的页地址流情况。根据这个页地址流才能找出当前要被替换的页面。显然，这样做是不现实的。因此，OPT 算法只是一种理想化的算法，然而它也是一种很有用的算法。实际上，经常把这种算法用来作为评价其他页面替换算法好坏的标准。在其他条件相同的情况下，哪一种页面替换算法的命中率与 OPT 算法最接近，那么它就是一种比较好的页面替换算法。

(5) 近期最少使用算法 (LFU 算法)。近期最少使用 (Least Frequently Used algorithm, LFU) 算法选择近期最少访问的页面作为被替换的页面。显然，这是一种非常合理的算法，因为到目前为止最少使用的页面，很可能也是将来最少访问的页面。该算法既充分利用了主存中页面调度情况的历史信息，又正确反映了程序的局部性。但是，这种算法实现起来非常困难，它要为每个页面设置一个很长的计数器，并且要选择一个固定的时钟为每个计数器定时计数。在选择被替换页面时，要从所有计数器中找出一个计数值最大的计数器。

10.3 用户编程接口

10.3.1 库函数与系统调用有什么不同

库函数调用是语言或应用程序的一部分，它是高层的，完全运行在用户空间，为程序员提供调用真正的在幕后完成实际事务的系统调用接口。而系统函数是内核提供给应用程序的接口，属于系统的一部分。函数库调用是语言或应用程序的一部分，而系统调用是操作系统的一部分。

库函数与系统调用的区别见表 10-2。

表 10-2 库函数与系统调用的区别

| 库 函 数 | 系 统 调 用 |
|-----------------------------------|------------------------------|
| 在所有的 ANSI C 编译器版本中，C 库函数是相同的 | 各个操作系统的系统调用是不同的 |
| 它调用函数库中的一段程序（或函数） | 它调用系统内核的服务 |
| 与用户程序相联系 | 是操作系统的一个入口点 |
| 在用户地址空间执行 | 在内核地址空间执行 |
| 它的运行时间属于“用户时间” | 它的运行属于“系统时间” |
| 属于过程调用，调用开销较小 | 需要在用户空间和内核上下文环境间切换，开销较大 |
| 在 C 库函数 libc 中有大约 300 个函数 | 在 UNIX 中有大约 90 个系统调用 |
| 典型的 C 函数库调用：system fprintf malloc | 典型的系统调用：chdir fork write brk |

库函数调用通常比行内展开的代码慢，因为它需要付出函数调用的开销。但系统调用比库函数调用还要慢很多，因为它需要把上下文环境切换到内核模式。

10.3.2 静态链接与动态链接有什么区别

静态链接是指把要调用的函数或者过程直接链接到可执行文件中，成为可执行文件的一部分。换句话说，函数和过程的代码就在程序的 exe 文件中，该文件包含了运行时所需的全部代码。静态链接的缺点是当多个程序都调用相同函数时，内存中就会存在这个函数的多个拷贝，这样就浪费了内存资源。

动态链接是相对于静态链接而言的，动态链接所调用的函数代码并没有被拷贝到应用程序的可执行文件中去，而是仅仅在其中加入了所调用函数的描述信息（往往是一些重定位信息）。仅当应用程序被装入内存开始运行时，在操作系统的管理下，才在应用程序与相应的动态链接库（dynamic link library, dll）之间建立链接关系。当要执行所调用 dll 中的函数时，根据链接产生的重定位信息，操作系统才转去执行 dll 中相应的函数代码。

静态链接的执行程序能够在其他同类操作系统的机器上直接运行。例如，一个 exe 文件是在 Windows 2000 系统上静态链接的，那么将该文件直接拷贝到另一台 Windows 2000 的机器上，是可以运行的。而动态链接的执行程序则不可以，除非把该 exe 文件所需的 dll 文件都一并拷贝过去，或者对方机器上也有所需的相同版本的 dll 文件，否则是不能保证正常运行的。

10.3.3 静态链接库与动态链接库有什么区别

静态链接库就是使用的 .lib 文件，库中的代码最后需要链接到可执行文件中去，所以静态链接的可执行文件一般比较大一些。

动态链接库是一个包含可由多个程序同时使用的代码和数据的库，它包含函数和数据的模块的集合。程序文件（如 .exe 文件或 .dll 文件）在运行时加载这些模块（也即所需的模块映射到调用进程的地址空间）。

静态链接库和动态链接库的相同点是它们都实现了代码的共享。不同点是静态链接库 lib 中的代码被包含在调用的 exe 文件中，该 lib 中不能再包含其他动态链接库或者静态链接库了。而动态链接库 dll 可以被调用的 exe 动态地“引用”和“卸载”，该 dll 中可以包含其他动态链接库或者静态链接库。

10.3.4 用户态和核心态有什么区别

核心态与用户态是操作系统的两种运行级别，它用于区分不同程序的不同权利。核心态就是拥有资源多的状态，或者说访问资源多的状态，也称之为特权态。相对来说，用户态就是非特权态，在此种状态下访问的资源将受到限制。如果一个程序运行在特权态，则该程序就可以访问计算机的任何资源，即它的资源访问权限不受限制。如果一个程序运行在用户态，则其资源需求将受到各种限制。例如，如果要访问操作系统的内核数据结构，如进程表，则需要在特权态下才能办到。如果要访问用户程序里的数据，则在用户态下就可以了。

Intel CPU 提供 Ring0~Ring3 4 种级别的运行模式。Ring0 级别最高，Ring3 最低。

用户态：Ring3 运行于用户态的代码则要受到处理器的诸多检查，它们只能访问映射其地址空间的页表项中规定的在用户态下可访问页面的虚拟地址，且只能对任务状态段（TSS）中 I/O 许可位图（I/O Permission Bitmap）中规定的可访问端口进行直接访问。

核心态：Ring0 在处理器的存储保护中，核心态或者特权态（与之相对应的是用户态）是操作系统内核所运行的模式。运行在该模式的代码，可以无限制地对系统存储、外部设备进行访问。

当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）。此时处理器处于特权级最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3级）用户代码中运行。

核心态下 CPU 可执行任何指令，在用户态下 CPU 只能执行非特权指令。当 CPU 处于核心态时，可以随意进入用户态；而当 CPU 处于用户态时，用户从用户态切换到核心态只有在系统调用和中断两种情况下发生。一般程序一开始都是运行于用户态，当程序需要使用系统资源时，就必须通过调用软中断进入核心态。

核心态和用户态各有优势：运行在核心态的程序可以访问的资源多，但可靠性、安全性要求高，维护管理都较复杂；用户态程序访问的资源受限，但可靠性、安全性要求低，自然编写维护起来都较简单。一个程序到底应该运行在核心态还是用户态取决于其对资源和效率的需求。

那么什么样的功能应该在核心态下实现呢？首先，CPU 管理和内存管理都应该在核心态实现。这些功能可不可以在用户态下实现呢？当然能，但是不太安全。就像一个国家的军队（CPU 和内存在计算机里的地位就相当于一个国家的军队的地位）交给老百姓来管一样，是非常危险的。所以从保障计算机安全的角度来说，CPU 和内存的管理必须在核心态实现。

诊断与测试程序也需要在核心态下实现，因为诊断和测试需要访问计算机的所有资源。输入输出管理也一样，因为要访问各种设备和底层数据结构，也必须在核心态实现。

对于文件系统来说，则可以一部分放在用户态，一部分放在核心态。文件系统本身的管理，即文件系统的宏数据部分的管理，必须放在核心态，不然任何人都可能破坏文件系统的结构；而用户数据的管理，则可以放在用户态。编译器、网络管理的部分功能、编辑器用户程序，自然都可以放在用户态下执行。

10.3.5 用户栈与内核栈有什么区别

内核在创建进程的时候，在创建 `task_struct` 的同时，会为进程创建相应的堆栈。每个进程会有两个栈，一个用户栈，存在于用户空间；一个内核栈，存在于内核空间。当进程在用户空间运行时，CPU 堆栈指针寄存器里面的内容是用户堆栈地址，使用用户栈；当进程在内核空间时，CPU 堆栈指针寄存器里面的内容是内核栈空间地址，使用内核栈。

当进程因为中断或者系统调用而陷入内核态时，进程所使用的堆栈也要从用户栈转到内核栈。进程陷入内核态后，先把用户态堆栈的地址保存在内核栈之中，然后设置堆栈指针寄存器的内容为内核栈的地址，这样就完成了用户栈向内核栈的转换；当进程从内核态恢复到用户态之后时，在内核态之后的最后将保存在内核栈里面的用户栈的地址恢复到堆栈指针寄存器即可。这样就实现了内核栈和用户栈的互转。

那么，知道从内核转到用户态时用户栈的地址是在陷入内核的时候保存在内核栈里面的，但是在陷入内核的时候，如何知道内核栈的地址？关键在进程从用户态转到内核态的时候，进程的内核栈总是空的。这是因为当进程在用户态运行时，使用的是用户栈，当进程陷入到内核态时，内核栈保存进程在内核态运行的相关信息，但是一旦进程返回到用户态后，内核栈中保存的信息无效，会全部恢复，因此每次进程从用户态陷入内核的时候得到的内核栈都是空的，所以在进程陷入内核的时候，直接把内核栈的栈顶地址给堆栈指针寄存器就可以了。

尽管程序员领着一份不错的薪水，可是他们也同样付出了巨大的精力与时间。随着软件规模的日益庞大，用户需求的不确定以及快速变更，使得软件开发已经不能停留在小作坊式的个人英雄时代，它已经发展为如今的依赖团队合作的行为，常规的管理方法已经无法满足软件开发的实际需求。而软件工程正是研究如何以系统性的、规范化的、可量化的过程化方法高效开发与管理、维护软件的交叉性学科，通过软件工程来降低程序员的烦恼。考查软件工程相关知识是衡量程序员专业知识水平的重要依据。

11.1 软件工程过程与方法

11.1.1 软件工程过程有哪些

软件工程是一门系统科学，是研究与应用如何以系统性的、规范性的、可量化的过程化方法来开发与维护软件，以及如何把经过时间考验而证明正确的管理技术和当前能够得到的最好的技术方法结合起来的方法。

标准的软件开发过程具备一定的流程，一般包括以下几个方面的内容：可行性分析、需求分析、设计、编码与实现、测试以及运行与维护。

(1) 可行性分析。

可行性分析是系统在正式立项之前必须进行的一项工作，它的目的不是为了分析软件开发过程中的问题，也不是为了解决软件开发过程中可能存在的问题，而是确定软件系统是否有价值做、是否能够以尽可能小的代价在尽可能短的时间内解决问题。

具体而言，在可行性分析阶段，要确定软件的开发目标与总的要求，所以在做可行性分析的时候，一般需要考虑技术是否可行、经济效益是否可行、用户操作是否可行、法律与社会是否可行等。例如，对于一个超市商品价格查询系统而言，就需要调查顾客是否希望使用这样的软件，超市商品价格来源是哪里？技术上是否能够实现等？

可行性分析一般都由战略专家执行，该阶段的文档成果为《可行性分析报告》。

(2) 需求分析。

需求分析是软件项目成败的关键，它是回答客户做什么的问题，是一个对用户的需求进行正确加工、正确理解，然后把它用软件工程开发语言表达出来的过程。需求分析不仅仅是用户需求，也应该是开发中遇到的所有的需求。例如，在做需求分析时，首先需要弄清楚该项目的目的是为了解决什么问题；其次弄清楚测试案例中应该输入什么数据，最后就是弄清楚哪些人需要使用本系统等。

本阶段的基本任务是和用户一起确定要解决的问题，建立软件的逻辑模型，编写需求规格说明书文档并最终得到用户的认可。需求分析的主要方法有结构化分析方法、数据流程图和数据字典等方法。本阶段一般能形成软件需求说明书、数据要求说明书以及初步的用户手册。

需求分析需要领域专家与系统架构师都参与进来,阶段性文档成果为《需求分析说明书》等。

(3) 设计。

软件设计的主要任务是将软件分解成模块,即能实现某个功能的数据和程序说明、可执行程序的程序单元。这个模块可以是一个函数、过程、子程序、一段带有程序说明的独立的程序和数据,也可以是可组合、可分解以及可更换的功能单元。软件设计可以分为概要设计和详细设计两个阶段。概要设计就是结构设计,其主要目标就是给出软件的模块结构。在详细设计中,首先就是要设计出模块的程序流程、算法和数据结构,其次是设计数据库,常用方法还是结构化程序设计方法。

该阶段的文档成果有《概要设计说明书》、《业务用例文档》、《详细设计说明书》、《技术用例文档》等。

(4) 编码与实现。

编码是指把软件设计转换成计算机可以接受的程序,即写成以某一程序设计语言表示的“源程序清单”。程序设计语言可以是 C、C++、C#或 Java 等。当前软件开发中除在专用场合,已经很少使用 20 世纪 80 年代的高级语言了,取而代之的是面向对象程序设计语言,而且面向对象的开发语言和开发环境大都合为一体,大大提高了开发的速度。由面向对象的开发语言开发的项目,系统的可扩展性与可维护性都大大增强。

该阶段的文档成果有《接口文档》、《关键算法文档》等。

(5) 测试。

软件测试贯穿于软件开发的整个过程,它的目的是以较小的代价发现尽可能多的错误。要实现这个目标关键在于根据软件开发各阶段的规格说明和程序的内部结构精心设计一套出色的测试用例(测试数据和预期的输出结果组成了测试用例),利用这些测试用例进行测试,从而发现程序中存在的错误和 bug。不同的测试方法有不同的测试用例设计方法,常用的测试方法有白盒测试和黑盒测试。白盒测试是一种测试单元内部如何工作的方法,目的是通过检查软件内部的逻辑结构,对软件中逻辑路径进行覆盖的测试,而黑盒测试不考虑程序的内部结构与特性,只根据程序功能或程序的外部特性设计测试用例,这两种测试方法都是依据软件的功能或对软件的行为描述,发现软件的接口、功能和结构错误。

该阶段的文档成果有《单元测试报告》、《集成测试报告》、《系统测试报告》等。

(6) 运行与维护。

虽然系统交付给用户,安装运行了,但是任何一个系统都不是一开始就能完全满足实际的应用需求,一般在交付使用后,还需要不断地进行再开发。而维护是指在已完成对软件的研制(分析、设计、编码和测试)工作并交付使用以后,对软件产品所进行的一些软件工程的活。即根据软件运行的情况,对软件进行适当修改与维护,如完善性维护、适应性维护,以适应新的要求,以及纠正运行中发现的错误,编写软件问题报告、软件修改报告。一个系统的质量的高低和系统的分析、设计有很大的关系,和系统的维护也有很大的关系。

需要注意的是,软件工程比较强调文档的重要性,所以每个阶段最好能够有文档保存,因为每个阶段建立在上一个阶段的基础之上,如果基础出了问题,后续阶段都可能会出现相应的问题,文档正好可以备查。

11.1.2 常见的软件开发过程模型有哪些

软件开发过程描述的是软件构造、部署以及维护的一种方法,它规定了完成各项任务所需

的步骤。建立软件开发过程模型的理论基础是软件生命周期理论和相关的软件工程原则。

软件开发过程的核心思想是将软件开发过程分为若干个阶段，每个阶段都遵循“高内聚、低耦合”的特性，这样可以有助于简化问题、有助于验证分阶段的工作成功，并且加强对软件开发的管理。

常见的软件开发过程模型有瀑布模型、螺旋模型、增量模型、原型模型和 RUP 模型。

(1) 瀑布模型。

瀑布模型将软件生命周期划分为可行性分析、需求分析、设计、代码实现、测试、安装与维护等 6 个基本活动，并且规定了它们自上而下的固定次序，形如瀑布流水，逐层下落。图 11-1 所示为瀑布模型结构图。

在瀑布模型中，软件开发的各项活动都严格按照线性方式进行，当前活动接受上一项活动的工作结果，实施完成所需的工作内容。当前活动的工作结果需要进行验证，如果验证通过，则该结果作为下一项活动的输入，继续进行下一项活动，否则返回修改。

作为一种最早出现的软件开发模型，瀑布模型为项目提供了按阶段划分的检查点，当前一阶段完成后，开发人员只需要去关注后续阶段的工作，所以采用瀑布模型可以严格地保证软件产品最终能够按时交付使用。但是由于瀑布模型的这种线性关系，使得各个阶段的划分太过固定，阶段之间产生了大量的文档，从而会增加工作量；在瀑布模型中，用户只有等到整个开发过程的末期才能见到开发成果，从而会增加开发的风险；而且会产生一个严重的后果，就是早期的错误可能等到了测试才被发现，问题会被放大最终导致严重的后果。

(2) 螺旋模型。

螺旋模型是一种采用周期性的方法来进行软件系统开发的模型，它结合了瀑布模型与原型模型的特点，强调了其他模型所忽视的风险分析，常用于大型复杂系统的开发。图 11-2 所示为螺旋模型结构图。

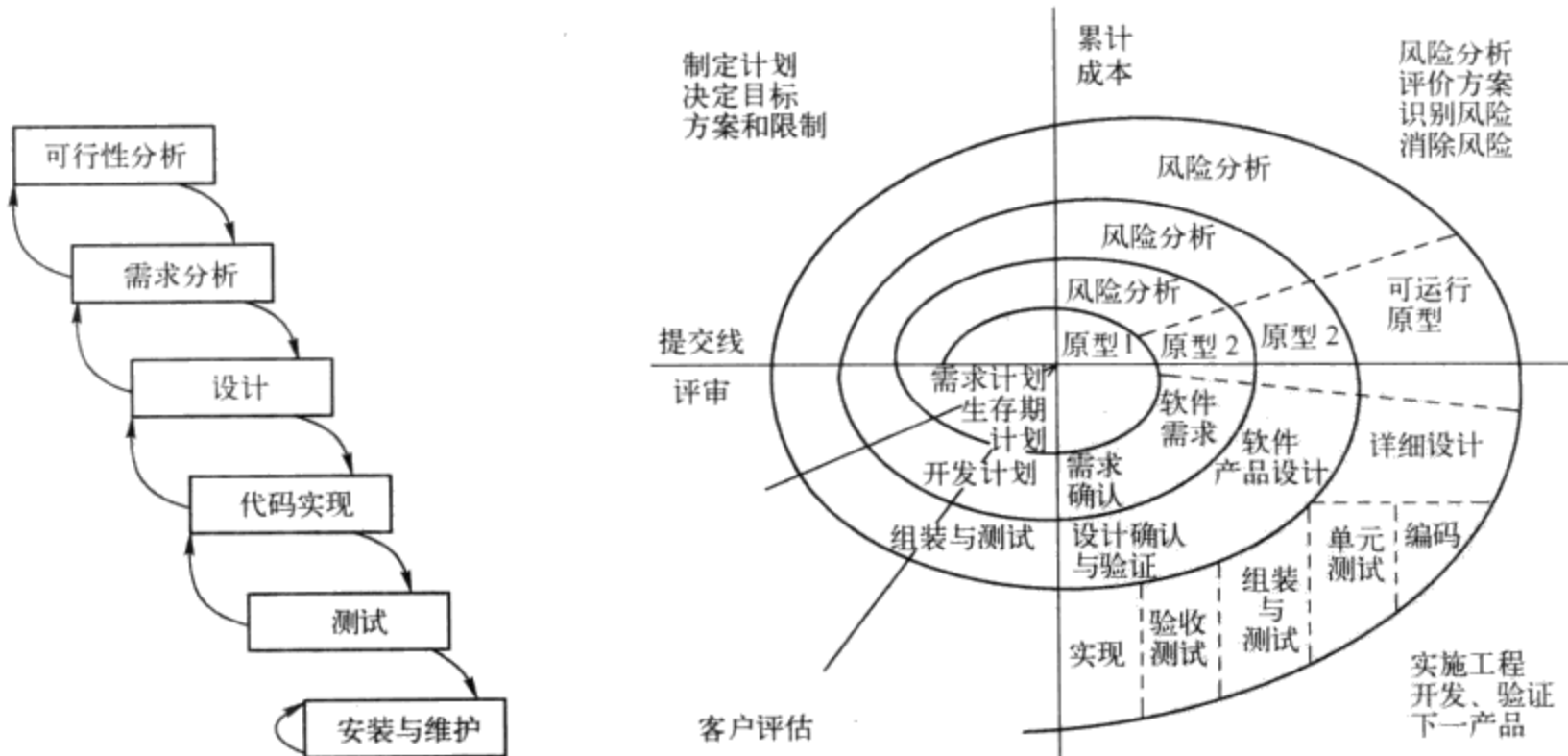


图 11-1 瀑布模型结构图

图 11-2 螺旋模型结构图

螺旋模型的优点主要表现在它设计上的灵活性，它可以在项目的各个阶段进行变更，以小的分段来构建大型系统可以使成本计算简单明了，而且风险分析可以使一些极端困难的问题和可能导致费用过高的问题被更改或取消，同时用户评价为需求的变更带来柔性，客户始终能够掌握项目的进展，从而提高需求的准确性与开发的高效性。而螺旋模型的缺点是由于它强调风

险分析, 要求客户接受并相信这种分析, 而做出相关反应是不容易的, 需要开发人员具有相当丰富的风险凭据经验和专门知识, 而且要求用户参与阶段评价, 对用户来说比较困难, 过多的迭代次数也会增加开发成本, 推迟软件交付的时间。

(3) 原型模型。

原型是指模拟某种产品的原始模型, 软件系统的原型是软件系统的一个早期可以运行的版本。原型模型是增量模型的一种形式, 在开发真实系统前, 首先需要构建一个简单的系统原型, 实现客户或未来的用户与系统的交互, 客户或用户在对原型进行使用的过程中, 不断发现问题, 从而达到进一步细化系统需求的目的。开发人员在已有原型的基础上, 通过逐步调整原型来满足客户或用户的需求, 确定客户的真正需求, 进而开发出客户或用户满意的系统。图 11-3 所示为原型模型结构图。

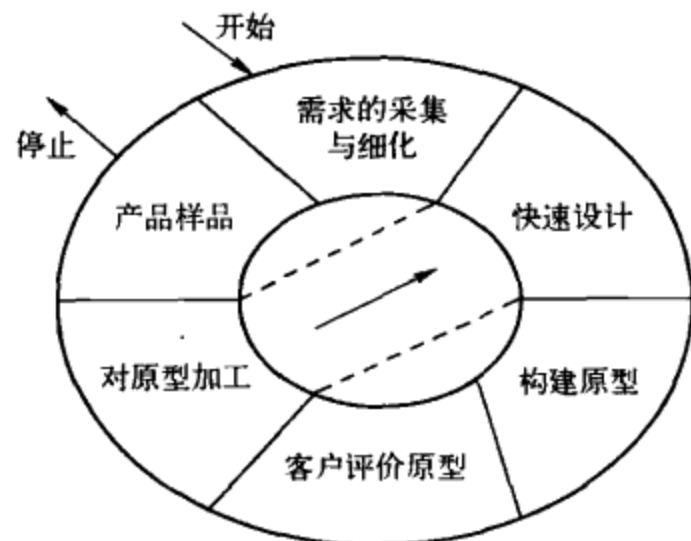


图 11-3 原型模型结构图

原型模型可以克服瀑布模型的缺点, 减少因为软件需求不明造成的开发风险。它的关键之处在于尽可能快速地建造出软件原型, 一旦确定了客户的真正需求, 所建造的原型将被丢弃。因此, 使用原型模型进行软件开发, 最重要的是必须迅速建立原型, 随之迅速修改原型, 以反映客户的需求, 而不是系统的内部结构。

(4) 增量模型。

增量模型也称为渐增模型, 是在项目的开发过程中以一系列的增量方式开发系统。在增量模型中, 软件被作为一系列的构件来设计、实现、集成与测试。每一个构件都由多种相互作用的模块所形成的提供特定功能的代码片段构成。在增量模型开发中, 将整个产品分解成若干个构件, 每次并不是交付一个可运行的完整产品, 而是交付系统的部分可运行子系统。此种方法的优点是可以较好地适应客户或用户需求的变化, 客户或用户可以分批不间断地看到运行良好的子系统, 从而降低开发风险。图 11-4 所示为增量模型结构图。

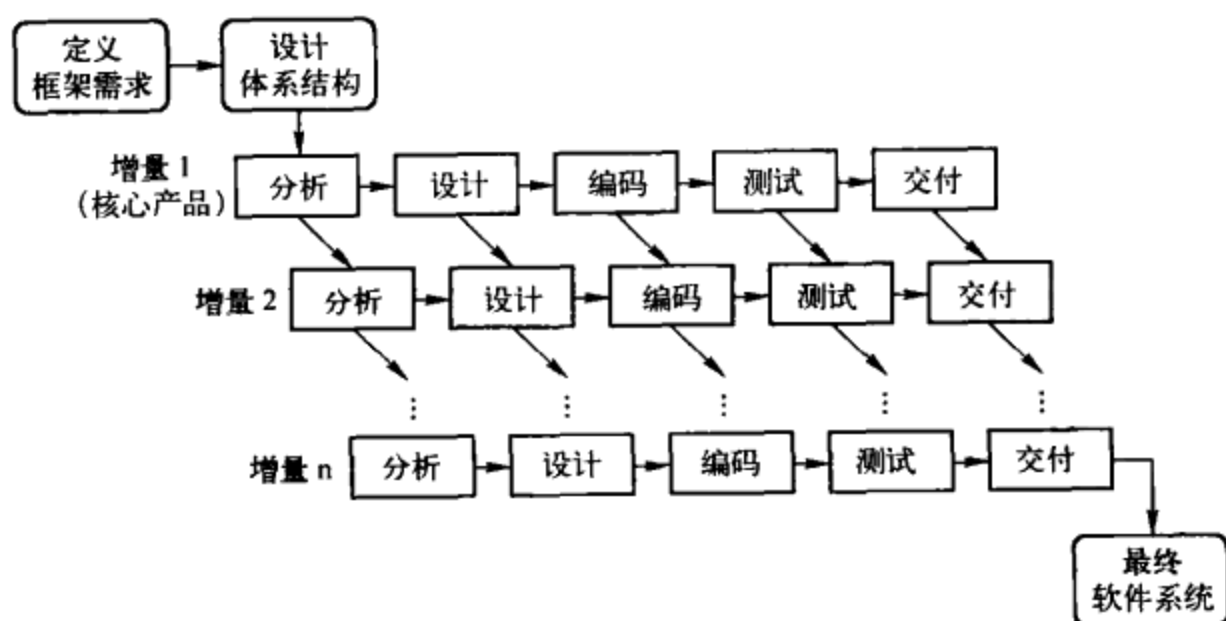


图 11-4 增量模型结构图

尽管如此, 增量模型也存在着诸多缺陷: 首先, 由于系统是进行增量开发的, 所以每次开发的构件在添加入系统的时候, 都有可能破坏已构建好的系统部分。其次, 需求的变化是不可避免的, 增量模型的灵活性可以使其适应这种变化的能力大大优于瀑布模型和快速原型模型,

但也容易使软件过程的控制失去整体性。

在增量模型中，第一个增量往往是实现基本需求的核心产品。核心产品交付用户使用后，经过评价形成下一个增量的开发计划，它包括对核心产品的修改和一些新功能的发布。这个过程在每个增量发布后不断重复，直到产生最终的完善产品。例如，使用增量模型开发字处理软件。可以考虑，第一个增量发布基本的文件管理、编辑和文档生成功能，第二个增量发布更加完善的编辑和文档生成功能，第三个增量实现拼写和文法检查功能，第四个增量完成高级的页面布局功能。

(5) 统一软件过程 RUP 模型。

RUP (Rational Unified Process, Rational 统一过程) 是 Rational 公司提出的一套软件开发过程模型。它将用户需求转化为软件系统所需活动的集合，描述了一系列相关的软件工作流程，它们具有相同的结构，即相同的流程框架。

RUP 吸收了多种开发模型的优点，具有良好的操作性与实用性。统一过程不仅是一个简单的过程，而且是一个通用的过程框架。它可以用于各种不同类型的软件系统、各种不同的应用领域、各种不同的项目规模等。

RUP 可以用二维坐标来描述。横轴通过时间组织，是过程展开的生命周期特征，体现开发过程的动态结构，用来描述它的术语主要包括周期 (Cycle)、阶段 (Phase)、迭代 (Iteration) 和里程碑 (Milestone)；纵轴以内容来组织为自然的逻辑活动，体现开发过程的静态结构，用来描述它的术语主要包括活动 (Activity)、产物 (Artifact)、工作者 (Worker) 和工作流 (Workflow)。

RUP 中的软件生命周期在时间上被分解为 4 个阶段，即初始阶段 (Inception)、细化阶段 (Elaboration)、构建阶段 (Construction) 以及交付阶段 (Transition)。每个阶段结束于一个主要的里程碑，而每个阶段本质上是两个里程碑之间的时间跨度。图 11-5 RUP 所示为 RUP 工作流程示意图。具体而言，四个阶段的功能如下所示

- 初始阶段：初始阶段的目标是为系统建立商业案例并确定项目的边界，在这个阶段定义了最终产品视图、商业模型并确定系统范围。它以需求分析为主，建立系统整体结构。初始阶段结束时的第一个重要的里程碑是生命周期目标里程碑，它用于评价项目基本的生存能力。
- 细化阶段：设计及确定系统的体系结构，制定工作计划及资源要求。针对第一阶段需求分析结果，进行设计、代码实现、测试、然后再反馈到需求分析。该阶段结束时第二个重要的里程碑是生命周期结构里程碑，它为系统的结构建立了管理基准并使项目能够在构建阶段中进行衡量。
- 构建阶段：构建产品并继续演进需求、体系结构、计划直至产品提交。对初始阶段的需求进行设计、代码实现、测试、反馈。重复需求、设计、编程、测试的过程。构建阶段的里程碑是初始功能里程碑，它决定了产品是否可以在测试环境中进行部署。

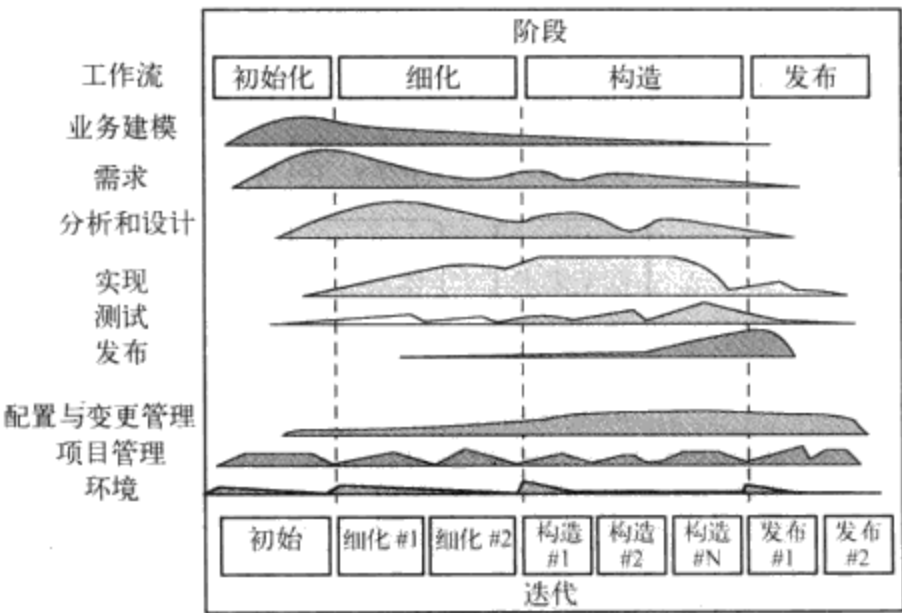


图 11-5 RUP 工作流程示意图

- 交付阶段：把产品提交给用户使用，综合测试，交付可运行产品。该阶段结束时的里程碑是产品发布里程碑。

RUP 中的每个阶段都可以进一步分解，进行迭代，每一个迭代都是一个完整的开发循环，产生一个可运行的软件版本，通过增量式开发，反复迭代，最终形成交付用户使用的完整产品。图 11-6 所示为迭代过程图。

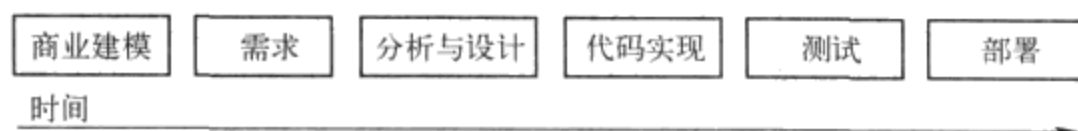


图 11-6 迭代过程图

采用 RUP 模型进行软件开发，可以提高团队的生产力，同时它建立了清晰的过程结构，为软件开发过程提供了较大的通用性，但由于它只是一个软件开发过程，没有覆盖软件过程的全部内容（例如，它缺少关于软件运行与支持等方面的内容，而且没有支持多项目的开发结构），所以，在开发组织内大范围实现重用的可能性就降低了。

不同的软件过程模型对软件开发过程有不同的理解和认识，支持不同的软件项目和开发组织。表 11-1 对比和分析了各个软件过程模型的特点及其适用的软件项目类型。

表 11-1 各模型对比

| 模 型 | 特 点 | 适 用 范 围 |
|--------|--------------------------------|-----------------------------------|
| 瀑布模型 | 分阶段，各个阶段完成后都有评审，允许反馈，要求预先确定需求 | 需求定义完善、不易变更的系统 |
| 螺旋模型 | 具有瀑布模型、快速原型模型的特点，并引进了风险分析活动 | 需求难以获取和确定、开发风险较大的系统 |
| 原型模型 | 不要求需求预先完备定义，支持用户参与，能够适应用户需求的变化 | 需求复杂、动态变化、难以确定的系统 |
| 增量模型 | 增量式开发，允许开发活动并行 | 技术风险较大、用户需求较为稳定的系统 |
| RUP 模型 | 可改造、扩展和剪裁，可以对它进行设计、开发、维护 | 复杂、需求不确定的系统，软件开发项目组拥有丰富的软件开发和管理经验 |

各种开发模型各有特色，开发人员可根据项目的不同特点，选择不同的开发模型。

11.1.3 什么是敏捷开发

敏捷开发是一种新型的软件开发方法，它以人为核心、迭代、循序渐进地进行软件开发，它具有应对快速需求变化的能力。在敏捷开发中，软件项目的构建被切分成多个子项目，各个子项目的成果都经过测试，具备集成和可运行的特征。具体而言，敏捷开发就是把一个大项目分为多个相互联系，但也可独立运行的小项目，并分别完成，在此过程中软件一直处于可使用状态。

相比较“非敏捷软件开发”，敏捷开发更加强调团队合作以及业务专家之间的紧密协作、面对面的交流与沟通、频繁交付新的软件发布版本、紧凑而自我组织型的团队、能够很好地适应需求变化的代码编写和团队组织方法，而且它也更注重软件开发中人的作用。

敏捷开发是一项系统工程，主要有以下 12 条敏捷原则。

(1) 最优先要做的是通过尽早的、持续的交付有价值的软件来使客户满意。规划迭代故事时必须按照优先级安排，为客户先提供最有价值的功能。通过频繁迭代能与客户形成早期的良好合作，及时反馈提高产品质量。敏捷开发小组关注的是完成和交付具有用户价值的功能，而不是孤立的任务。以前都用需求规格说明书或者用例来编写详细的需求，敏捷开发使用用户故

事来罗列需求。使用基于用户故事的需求分析方法时，仍可能需要原型和编写文档，只是工作重点更多地转移到了口头交流。

(2) 即使到了系统开发的后期，也欢迎改变需求。敏捷开发过程利用变化来为客户创造竞争优势。敏捷开发过程参与者不怕变化，他们认为改变需求是好事情，因为这些改变意味着更了解市场的需求。

(3) 经常性地交付可以工作的软件，交付的时间间隔可以从几周到几个月不等，当然交付的时间间隔越短越好。迭代是受实践框限制的，意味着即使放弃一些功能也必须按时结束迭代。只要可以保证交付的软件可以很好地工作，那么交付时间越短，与客户协作就越紧密，对产品质量就更有益。虽然经过多次迭代，但并不是每次迭代的结果都需要交付给用户，敏捷开发的目标是让它们可以交付，这意味着开发小组在每次迭代中都会增加一些功能，增加的每个功能都是经过编码、测试，达到了可发布的质量标准的。

(4) 在整个项目开发期间，业务人员和开发人员必须天天都在一起工作，从而保证在项目开发过程中出现问题能够及时解决问题。软件项目不会依照之前设定的计划按部就班地执行，中间对业务的理解、软件的解决方案肯定会存在偏差，所以客户、需求人员、开发人员以及客户之间必须进行有意义的、频繁的交互，这样就可以在早期及时地发现问题并解决问题。

(5) 激励个人来构建项目。给他们提供所需要的环境和支持，并且信任他们能够完成工作。业务和技术是引起不确定的两个主要方面，人是第三个方面，而业务和技术又必须由人来执行，所以能够激励人来解决这些问题是解决不确定性的关键。只要个人的目标和团队的目标一致，就需要鼓舞起每个人的积极性，以个人为中心构建项目，提供所需的环境、支持与信任。

(6) 采用面对面的交谈方式，因为在团队内部，最具有效果并且富有效率的传递信息的方法就是面对面的交谈。在十几或者二十几个人组成的大团队中，文档是一种比较合适的传递知识和交流的途径。而敏捷团队一般不会很多人（大团队实施敏捷时也会分成多个小的敏捷团队），所以大量的文档交流其实并不是非常经济的做法。此时面对面的交谈反而更快速有效。

(7) 可以工作的软件是首要的进度度量标准。一般的工作都比较容易衡量任务进展，对于软件来说，在软件没有编码、测试、客户认可之前，都不能因为代码编写了多少行，测试用例运行了多少个就去度量这个功能是否完成了。衡量这个功能是否完成的首要标准就是这个功能可以工作了，对用户来说已经可以应用了。

(8) 敏捷过程提要保持可持续的开发速度。责任人、开发者和用户应该能够保持一个长期的、恒定的开发速度。敏捷过程希望能够可持续地进行开发，开发速度不会随着迭代的任务不同而不同，不欣赏所谓的拼一拼也能完成的态度，开发工作不应该是突击行为。

(9) 不断地关注优秀的技能和好的设计会增强敏捷能力。敏捷过程有很多好的技术实践可以加强产品敏捷能力，很多原则、模式和实践也可以增强敏捷开发能力。

(10) 简单。使未完成的工作最大化的艺术是根本的。不可能预期后面需求会如何变化，所以不可能一开始就构建一个完美的架构来适应以后的所有变化。敏捷开发团队不会去构建明天的软件，而把注意力放在如何通过最简单的方法完成现在需要解决的问题。

(11) 最好的构架、需求和设计出自于自组织的团队。敏捷中有很多种实践，迭代式开发是主要的实践方法，而自组织团队也是主要的实践之一。在自组织团队中，管理者不再发号施令，而是让团队自身寻找最佳的工作方式来完成工作。

(12) 每隔一定时间，团队会在如何才能更有效地工作方面进行反省，然后相应地对自己的行为进行调整。由于很多不确定性因素会导致计划失效，如项目成员增减、技术应用效果、用户需求的改变、竞争者对团队的影响等都会让团队作出不同的反应。敏捷不是基于预定义的

工作方式，而是基于经验性的方式，对以上这些变化，小组通过不断的反省调整来保持团队的敏捷性。

敏捷方法有时被误认为是无计划性与纪律性的方式，其实是不准确的。虽然敏捷方法有其特殊之处，但是它与其他方法也存在着共性，如迭代开发，关注互动沟通，减少中介过程的无谓资源消耗。

常见的敏捷开发方法有自适应软件开发（Adaptive Software Development, ASD）、水晶方法（Crystal Method）、特性驱动开发（Feature Driven Development, FDD）、动态系统开发方法（Dynamic Systems Development Method, DSDM）与极限编程（Extreme Programming, XP）等。

11.1.4 UML 中一般有哪些图

UML 提供了 8 种图，它们分别是类图、对象图、用例图、交互图（包括序列图和协作图）、状态图、活动图、构件图和部署图。其中类图和对象图属于静态图，状态图和活动图属于行为图，序列图和协作图属于交互图，构件图和部署图属于实现图。

（1）类图。类图展现了一组对象、接口、协作以及它们之间的关系。类图用于对系统的静态设计视图建模，包括对系统的词汇建模，对简单的协作建模，对逻辑数据库模式建模。

（2）对象图。对象图展现了一组对象以及它们之间的关系。当对系统的静态设计视图或者静态进程视图建模时，主要是使用对象图对对象结构进行建模。

（3）用例图。用例图展现了一组用例、参与者（actor）以及它们之间的关系。用例图用于对系统的静态用例视图进行建模。

（4）交互图。此种图包括序列图和协作图。它们用于对系统的动态交互方面进行建模。

（5）状态图。状态图展现了一个状态机，它由状态、转换、事件和活动组成。状态图关注系统的动态视图，它对于接口、类和协作的行为建模尤为重要，它强调对象行为的时间顺序。

（6）活动图。活动图是一种特殊的状态图，它展现了在系统内从一个活动到另一个活动的流程。活动图专注于系统的动态视图。

（7）构件图。构件图展现了一组构件之间的组织和依赖。构件图专注于系统的静态实现视图。它与类图相关，通常把构件映射为一个或者多个类、接口或协作。

（8）部署图。部署图展现了运行处理结点以及其中的构件的配置。部署图给出了体系结构的静态实施视图。

11.2 软件工程思想

11.2.1 什么是软件配置管理

由于软件系统的日益庞大，参与软件开发的团队人员越来越多，软件开发遇到了越来越多的问题，如发布版本错误、异地不能正常工作、安装后无法正常工作、已经解决的缺陷过后又出现错误、找不到最新修改了的源程序、开发人员未经授权修改代码或文档、人员流失导致技术泄密、无法重现历史版本等。软件配置管理就是用于解决如何保证产品的精确、如何重建先前发布的产品的有效方法。

软件配置管理（Software Configuration Management, SCM）又称为软件形态管理或软件建构管理。它是一种标识、组织和控制修改的技术，用于界定软件的组成项目，对每个项目的

变更进行管控，并维护不同项目之间的版本关联，以使软件在开发过程中任一时间的内容都可以被追溯。软件配置管理贯穿于整个软件生命周期，它为软件开发提供了一整套管理办法和活动原则，无论是对于软件企业管理人员还是研发人员都有着重要的意义。

软件配置管理的好坏将直接影响软件产品的质量，而实施有效的软件配置管理，具有以下6个方面的优势：

(1) 解决了由于开发经费及开发时间的限制所带来的问题使得许多问题得到阶段性的处理。从而减少了软件产品的不断升级。

(2) 使得开发商开发过程有了规范化的管理，用户不必再投入大量的经费去开发新产品，节省了大量的人力、物力和时间。

(3) 在软件的团队式开发中，对人员流动进行了有效地管理，使得软件开发在人员方面减少出错的概率。

(4) 避免了没经测试的软件加入到产品中，提高了产品的质量。

(5) 使得用户与开发商之间有了有效的沟通手段，用户的利益得到有效保证。

(6) 使得软件生产规模化，使得企业生产出软件企业内部的软件标准构件仓库成为可能，应用软件产品上升到一种高水平、避免重复开发的状态，开发时间得到了保证，而且成本也相应降低，使产品更加具有市场竞争力。

对软件进行配置管理，一般是依靠工具来实现的，常见的软件配置管理工具有 Microsoft 公司的 SourceSafe、CVS (Concurrent Version System, 并行版本系统)、Rational 公司的 ClearCase 等。

11.2.2 什么是 CMMI

CMMI (Capability Maturity Model Integration, 软件能力成熟度模型集成) 是由美国国防部与卡内基梅隆大学和美国国防工业协会共同开发和研制的一项标准，其目的是帮助软件企业对软件工程过程进行管理和改进，增强开发与改进能力，从而能按时地、不超预算地开发出高质量的软件。

该模型基本上覆盖了产品研发的各个过程领域，包括项目管理、需求、设计、开发、验证、确认、配置管理、质量保证、决策分析以及对研发的改进和培训等一系列活动。该模型按照成熟度等级的逐步提高，产品开发企业的产品研发风险越来越低，研发效率和质量越来越高。

信息系统和软件的 CMM 框架用来帮助组织改善其系统开发过程的成熟度，CMMI 一般分为 5 个级别，从第一级到第五级分别是初始级、可重复级、定义级、管理级和优化级，如图 11-7 所示。

(1) 初始级。

这一级有时称为无政府状态 (anarchy) 或混乱状态 (chaos)，软件过程是未加定义的随意过程，项目在执行时是随意甚至是混乱的，有些企业虽然指定了一些软件工程规范，但是如果这些规范不能够覆盖基本的关键过程要求，而且没有政策、资源等方面的保证时，那么它就被视为初始级。

(2) 可重复级。

在可重复级水平上，企业在项目实施上有资源准备，权责到人，能够遵守既定的计划与流程，对相关的项目实施人员有相应的培训，对整个流程有监测与控制，并与上级单位一起对项目与流程进行审查。企业在二级水平上体现了对项目的一系列的管理程序，这一系列的管理手段排除了企业在一小时完成任务的随机性，保证了企业的所有项目实施都会得到成功。

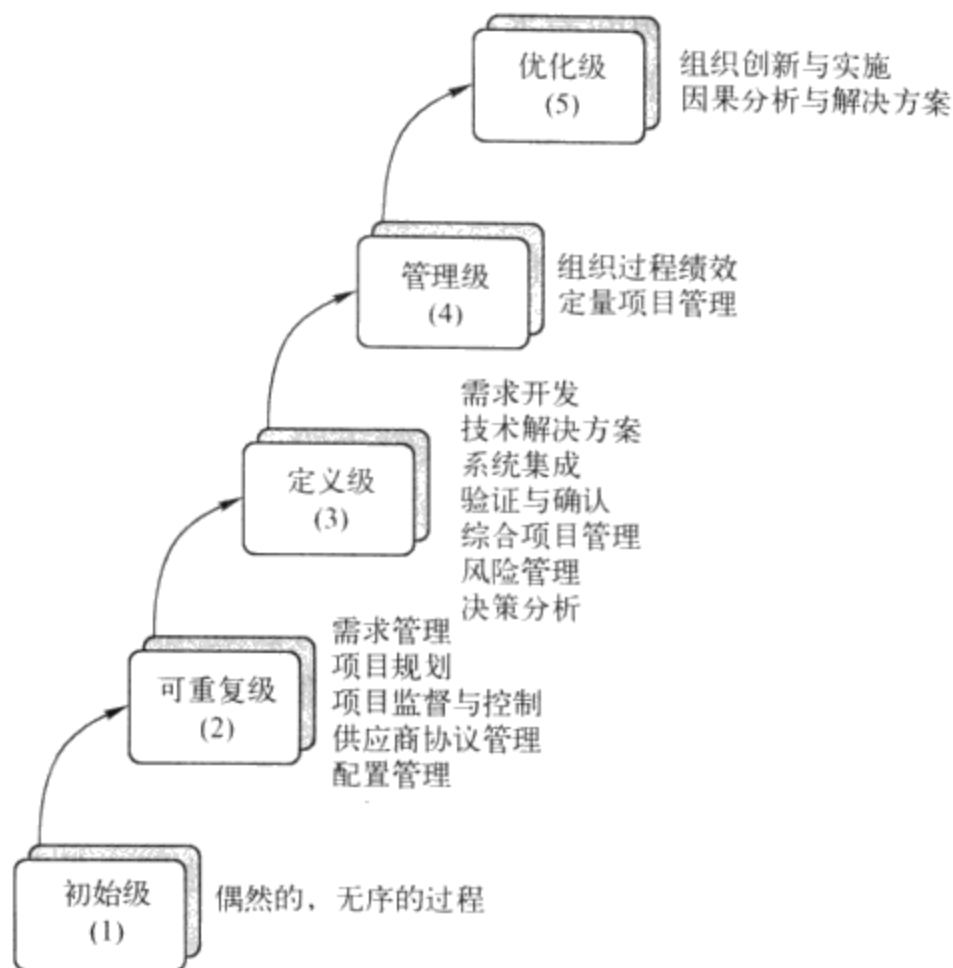


图 11-7 CMMI 5 级图

（3）定义级。

在定义级水平上，企业不仅能够对项目的实施有一整套的管理措施，保障了项目的完成，而且企业能够根据自身的特殊情况以及自己的标准流程，将这套管理体系与流程予以制度化。这样，企业不仅能够将同类的项目上升到成功的实施，而且在不同类的项目上也同样能够得到成功的实施。

（4）管理级。

在管理级水平上，企业的项目管理在形成一整套制度的基础上还要实现数字化的管理。对管理流程要做到量化与数字化，通过量化技术来实现流程的稳定性，实现管理的精度，降低项目实施在质量上的波动。

（5）优化级。

在优化级水平上，企业不仅能够通过信息化手段与数字化手段来实现对项目的管理，而且能够充分利用信息资料，对企业在项目实施的过程中可能出现的次品予以预防。主动地改善了流程，运用了新的技术，实现了流程的优化。如果企业达到了第五级，就表明该企业能够根据实际的项目性质、技术等因素，不断调整软件生产过程以求达到最佳。

CMMI 是一套行之有效的软件开发过程控制的规范，但它不是软件企业的目的，只是软件企业达到企业目的的一种有效手段。

11.2.3 如何提高软件质量

软件质量是指一个实体（产品或服务）的所有特性，基于这些特性可以满足客户明显的或隐含的需要，它是软件的灵魂。

软件质量不是一个纯粹的技术问题，而是一个系统的工程性问题。具体而言，它一般包括以下 3 个方面的内容：第一，符合目标，目标由客户指定，符合目标就是判断开发人员是不是在做需要做的事情。第二，符合需求，软件产品是不是在做客户让它做的事情。第三，符合实

实际需求，实际的需求包括用户明确说明的需求，包括隐含需求。

由于软件规模的日益庞大，内容的日益复杂，导致软件的质量问题变得越来越突出。软件危机的出现，产生了很多提高软件质量的理论与方法，如面向对象技术的提出、自动化工具（CASE 工具、过程控制软件）的应用、测试技术的提高、开发模型的完善等，但由于每一种方法都不是绝对的，还需要综合考虑各方面的情况。

为了提高软件质量，一般需要注意以下几个方面：

（1）要改进软件产品的质量首先应当从流程抓起，规范软件产品的开发过程。这是从根本上解决软件质量问题、提高软件开发效率的一个关键手段。无论做什么事情，都有一个循序渐进的过程，从计划到策略再到实现，软件流程都是按照这种思维来定义开发过程的。它根据不同的产品特点和以往的成功经验，定义从需求到最终产品交付的一整套流程。流程告诉开发人员该怎么一步一步去实现产品，可能会有哪些风险，如何去避免风险等。

（2）做好需求，需求是项目的灵魂。大部分的错误都是在软件开发前期引入的，不明确或者模糊的需求可能会带来不可避免的后果。如果这种情况发生在项目开发的初始阶段，最终很有可能导致软件返工或者失败。所以，可以通过提高需求分析和设计方面的技术，如原型法技术、分析模式、设计模式、面向对象设计、UML 等，来有效地提高软件质量。

（3）做好设计，设计最能体现一个项目团队的能力与水平。设计是把需求转换成系统的一个关键步骤，它需要从自然语言描述的需求中寻找出设计的基础单元，构建出整个系统的构架。一个好的设计基本上决定了产品的最终质量。

（4）加强规范，开发的过程中加强编程规范工作，并且实施软件配置管理，加强版本控制，都能有效地提高软件开发的效率与质量。

（5）全面质量控制要求在过程的每个阶段每个步骤上都要进行严格的验证与确认活动。所谓验证就是要用数据证明最终是不是在正确地制造产品；所谓确认就是要用数据证明是不是制造了正确的产品。

（6）加强文档化工作，文档是经验的保留。一个合格的软件产品除了代码合格以外，其实更重要的还应该具有完善的文档记录。对于一个企业而言，要想获得长期的发展，必须加强文档化工作。

（7）开展走读、评审和检视活动，尤其要加强代码走读，建议进行每日交叉走读活动。

（8）进行简单的度量分析活动。

（9）进行软件测试，软件测试是软件质量控制中的关键活动。软件测试的目的是要发现软件中的错误。一个好的测试是发现至今没有发现的错误。传统的软件测试专注于动态测试范畴，如单元测试、集成测试和系统测试。而测试工程的发展已经进入到了全流程的测试，包括开发过程前期的静态测试。软件测试是产品最终是否满足规定的需求或检测预期结果与实际结果之间差别的一种测试方法。

提高软件质量是整个软件研发团队的任务，各个项目组成员都应该为了这个目标做更多的工作。而且在软件的研发中，越早进行质量控制，越能保证软件的质量，而只有团队所有成员时时刻刻谨记软件质量的重要性，提高软件质量意识，才能保证最终交付给用户一个满意的软件产品。

在面试笔试中，面试官除了关注一些常见的计算机专业基础知识外，偶尔还会发散思维，考查求职者一些当前比较新颖或是平时不太被注意、使用的技术类题目，如云计算、物联网、设计模式等，以判断求职者是否及时准确地关注了当前技术发展的潮流，所以求职者对此类问题还是应该多些理解与认识。

12.1 设计模式

设计模式（Design Pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式的目的是为了代码重用，避免程序大量修改，同时使代码更容易被他人理解，并且保证代码的可靠性。显然，设计模式对自己、对他人、对系统都是有益的，设计模式使得代码编制真正的工程化，设计模式可以说是软件工程的基石。

GoF（Gang of Four）23 种经典设计模式如图 12-1 所示。

| | 创建型 | 结构型 | 行为型 |
|----|--|---|---|
| 类 | Factory Method（工厂方法） | Adapter_Class（适配器类） | Interpreter（解释器） Template Method（模板方法） |
| 对象 | Abstract Factory（抽象工厂） Builder（生成器） Prototype（原型） Singleton（单例） | Adapter_Object（适配器对象） Bridge（桥接） Composite（组合） Decorator（装饰） Façade（外观） Flyweight（享元） Proxy（代理） | Chain of Responsibility（职责链） Command（命令） Iterator（迭代器） Mediator（中介者） Memento（备忘录） Observer（观察者） State（状态） Strategy（策略） Visitor（访问者模式） |

图 12-1 23 种经典设计模式

常见的设计模式有工厂模式（Factory Pattern）、单例模式（Singleton Pattern）、适配器模式（Adapter Pattern）、享元模式（Flyweight Pattern）以及观察者模式（Observer Pattern）等。

12.1.1 什么是单例模式

在某些情况下，有些对象只需要一个就可以了，即每个类只需要一个实例。例如，一台计算机上可以连接多台打印机，但是这个计算机上的打印程序只能有一个，这里就可以通过单例模式来避免两个打印作业同时输出到打印机中，即在整个的打印过程中只有一个打印程序的实例。

简单来说，单例模式（也叫单件模式）的作用就是保证在整个应用程序的生命周期中，任何一个时刻，单例类的实例都只存在一个（当然也可以不存在）。单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例单例模式。单例模式只应在有真正的“单一实例”的需求时才可使用。

需要区分一下全局变量和单例模式。首先，全局变量是对一个对象的静态引用，全局变量确实可以提供单例模式实现的全局访问这个功能，但是它并不能保证应用程序中只有一个实

例，同时在编码规范中，也明确指出应该要少用全局变量，因为过多地使用全局变量，会造成代码难读，还有就是全局变量并不能实现继承（虽然单例模式在继承上也不能很好地处理，但是还是可以实现继承的）。而单例模式的话，其在类中保存了它的唯一实例，这个类可以保证只能创建一个实例，同时它还提供了一个访问该唯一实例的全局访问点。

使用单例模式，一般需要注意单例模式是用来实现在整个程序中只有一个实例的。单例类的构造函数必须为私有，同时单例类必须提供一个全局访问点。

12.1.2 什么是工厂模式

工厂模式专门负责实例化有大量公共接口的类。工厂模式可以动态地决定将哪一个类实例化，而不必事先知道每次要实例化哪一个类。客户类和工厂类是分开的。消费者无论什么时候需要某种产品，需要做的只是向工厂提出请求即可。消费者无需修改就可以接纳新产品。当然也存在缺点，就是当产品修改时，工厂类也要做相应的修改。

工厂模式包含以下几种形态：

（1）简单工厂（Simple Factory）模式。简单工厂模式的工厂类是根据提供给它的参数，返回的是几个可能产品中的一个类的实例，通常情况下它返回的类都有一个公共的父类和公共的方法。

（2）工厂方法（Factory Method）模式。工厂方法模式是类的创建模式，其用意是定义一个用于创建产品对象的工厂的接口，而将实际创建工作推迟到工厂接口的子类中。它属于简单工厂模式的进一步抽象和推广。多态的使用，使得工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。

（3）抽象工厂（Abstract Factory）模式。抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时使用的一种工厂模式，抽象工厂模式可以向客户端提供一个接口，使客户端在不指定产品的具体的情况下，创建多个产品族中的产品对象。根据 LSP 原则（即 Liskov 替换原则），任何接受父类型的地方，都应当能够接受子类型。因此，实际上系统所需要的，仅仅是类型与这些抽象产品角色相同的一些实例，而不是这些抽象产品的实例。换句话说，也就是这些抽象产品的具体子类的实例。工厂类负责创建抽象产品的具体子类的实例。

12.1.3 什么是适配器模式

适配器模式也称为变压器模式，它是把一个类的接口转换成客户端所期望的另一种接口，从而使原本因接口不匹配而无法一起工作的两个类能够一起工作。适配类可以根据所传递的参数返还一个合适的实例给客户端。

适配器模式主要应用于“希望复用一些现存的类，但是接口又与复用环境要求不一致的情况”，在遗留代码复用、类库迁移等方面非常有用。同时适配器模式有对象适配器和类适配器两种形式的实现结构，类适配器采用“多继承”的实现方式，会引起程序的高耦合，所以一般不推荐使用；而对象适配器采用“对象组合”的方式，耦合度低，应用范围更广。

例如，现在系统里已经实现了点、线、正方形，而现在客户要求实现一个圆形，一般的做法是建立一个 Circle 类来继承以后的 Shape 类，然后去实现对应的 display、fill、undisplay 方法。此时如果发现项目组其他人已经实现了一个画圆的类，但是他的方法名却和自己的不一样：displayhh、fillhh、undisplayhh，我们不能直接使用这个类，因为那样无法保证多态。而有的时候，也不能要求组件类改写方法名，此时可以采用适配器模式。

12.1.4 什么是享元模式

享元模式以共享的方式高效地支持大量的细粒度对象。享元模式能做到共享的关键是区分内蕴状态和外蕴状态，其中内蕴状态存储在享元内部，不会随环境的改变而有所不同，而外蕴状态是随环境的改变而改变的，外蕴状态不能影响内蕴状态，它们是相互独立的。将可以共享的状态和不可以共享的状态从常规类中区分开来，将不可以共享的状态从类里剔除出去。客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象。享元模式大幅度地降低内存中对象的数量。

12.1.5 什么是观察者模式

观察者模式（也被称为发布/订阅模式）提供了避免组件之间紧密耦合的另一种方法，它将观察者和被观察的对象分离开。在该模式中，一个对象通过添加一个方法（该方法允许另一个对象，即观察者注册自己）使本身变得可观察。当可观察的对象更改时，它会将消息发送到已注册的观察者。这些观察者使用该信息执行的操作与可观察的对象无关，结果是对象可以相互对话，而不必了解原因。Java 与 C# 的事件处理机制就是采用的此种设计模式。

例如，用户界面可以作为一个观察者，业务数据是被观察者，用户界面观察业务数据的变化，发现数据变化后，就显示在界面上。面向对象设计的一个原则是：系统中的每个类将重点放在某一个功能上，而不是其他方面。一个对象只做一件事情，并且将它做好。观察者模式在模块之间划定了清晰的界限，提高了应用程序的可维护性和重用性。

12.2 新技术

12.2.1 什么是云计算

云计算是分布式处理、并行处理和网络计算的发展，也是这些计算机科学概念的商业实现。它是一种基于互联网的计算方式，它将与 IT 相关的能力以服务的形式提供给用户，允许用户在不了解提供服务的技术、没有相关知识以及设备操作能力的情况下，通过 Internet 获取需要的服务。通过云计算技术，网络服务提供者可以在数秒之内，达成处理数以千万计的信息，达到和“超级计算机”同样强大效能的网络服务。

云计算的核心思想是将大量用网络连接的计算资源统一管理和调度，构成一个计算资源池向用户按需服务。提供资源的网络被称为“云”，所以云其实是网络、互联网的一种比喻说法而已。

云计算平台是一个强大的“云”网络，连接了大量并发的网络计算和服务，可利用虚拟化技术扩展每一个服务器的能力，将各自的资源通过云计算平台结合起来，提供超级计算和存储能力，通用的云计算体系结构图如图 12-2 所示。

从体系结构来看，云计算的底层由硬件组成，在此基础上分别是基础设施即服务（Infrastructure as a Service, IaaS）、平台即服务（Platform as a Service, PaaS）和软件即服务（Software as a Service, SaaS）。

（1）IaaS 是指以服务形式提供服务器、存储和网络硬件，这类设施一般是利用网络计算设施来构建虚拟化的环境，光纤、服务器、集群和动态配置软件被涵盖在 IaaS 之中。IaaS 有两种实现模式：公有的和私有的，如 Amazon EC2 在基础设施云中使用公共服务器池。私有化的服务则使用企业内部数据中心的一组公用或私有服务器池。

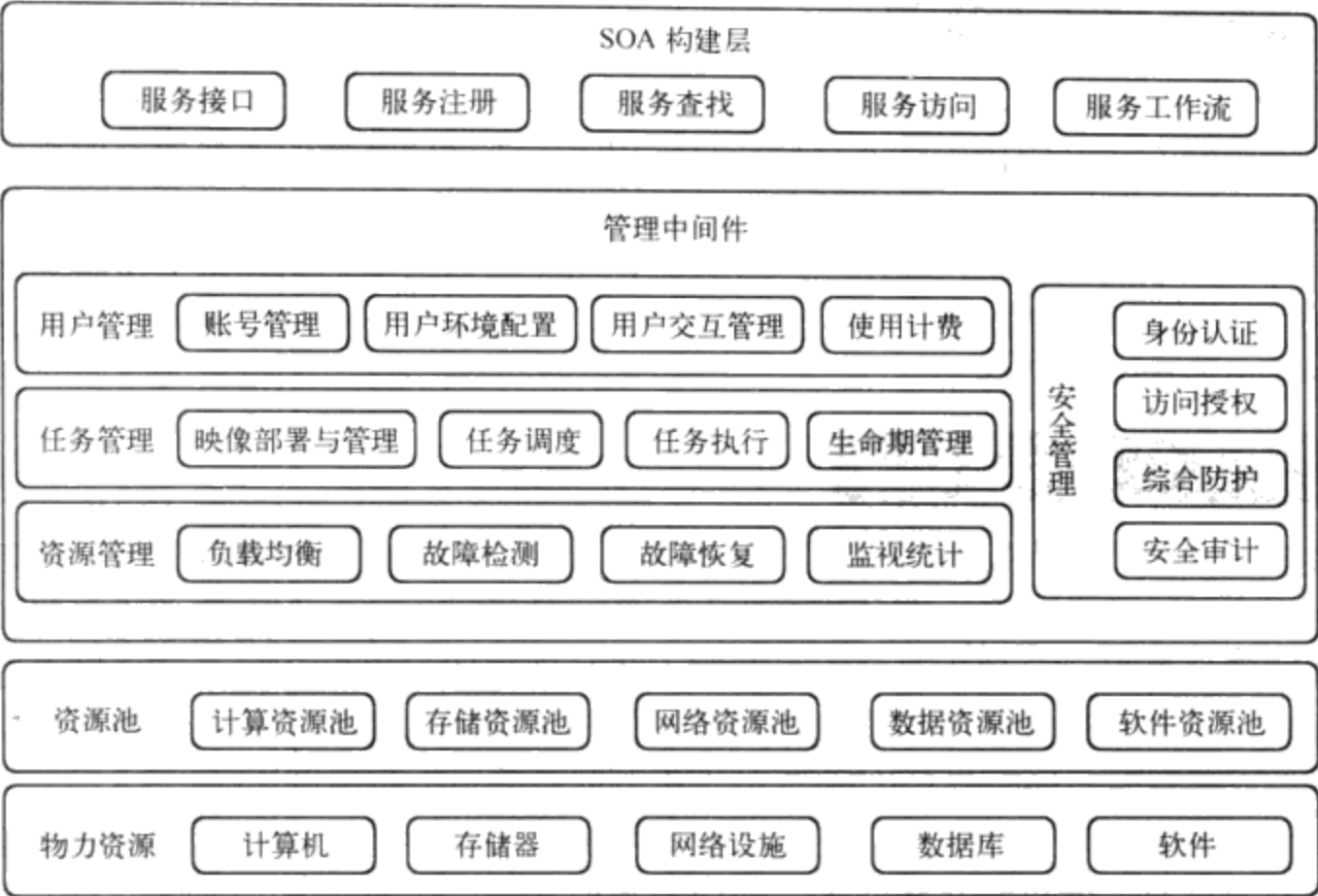


图 12-2 云计算体系结构图

(2) PaaS 是在 IaaS 之上的一层。它是指将软件研发的平台作为一种服务，以 SaaS 的模式提交给用户。因此，PaaS 也是 SaaS 模式的一种应用。但是，PaaS 的出现可以加快 SaaS 的发展，尤其是加快 SaaS 应用的开发速度。在云计算服务中，平台及服务包括以下类型服务：提供继承开发环境，集成 Web 服务和数据库，支持团队协作，提供使用设备。

(3) SaaS 是一种通过 Internet 提供软件的模式，用户无需购买软件，而是向提供商租用基于 Web 的软件来管理企业经营活动。这种类型的云计算通过浏览器把程序提供给成千上万的用户使用。SaaS 通常被应用于企业管理软件领域、产品技术和市场。

12.2.2 什么是物联网

物联网的概念由美国 MIT（麻省理工学院）的 Kevin Ashton 于 1999 年提出。它是指把所有物品通过各种信息传感设备，如传感器、射频识别技术、全球定位系统、红外感应器、激光扫描器、气体感应器等各种装置与技术，实时采集任何需要监控、连接、互动的物体或过程，采用其声、光、热、电、力学、化学、生物、位置等各种需要的信息，与互联网连接起来，实现智能化识别和管理。

现代社会发展在能源、交通、物流、金融等方面都遇到了发展的“瓶颈”，又在健康、医疗、服务等方面面对着人们的直接需求。选择信息技术，通过更智能的终端、覆盖更全面的网络、提供更良好的服务来解决经济发展、社会发展问题，提高人民生活水平，是人们对信息技术普遍信任的自然选择。

按照技术特征可以把物联网的业务分为 4 类：身份相关业务、信息汇聚型业务、协同感知类业务和泛在服务。

(1) 身份相关业务。

身份相关业务类的应用主要是利用射频标志（RFID）、二维码、条码等可以标志身份的技术，并基于身份所提供的各类服务。按照终端是去识别其他身份信息还是被识别，可以分为

主动模式和被动模式；按照服务是提供给个人还是提供给企业，又可以分为个人应用和企业业务两大类。

对于不同的应用实现的方式可能各有不同，但一般方法是在物上贴上 RFID 标签，读写设备通过读取 RFID 标签中的信息，尤其是 ID 信息，通过这个 ID 信息向物联网名称解析服务器请求，以获取该 ID 所对应的进一步详细信息的统一资源标志符（URI），读写设备通过这个统一资源标志符进行进一步的信息获取。

（2）信息汇聚型业务。

信息汇聚型业务主要是有物联网终端采集、处理、经通信网络上报数据，由物联网平台处理，提交给具体的应用和服务，由物联网平台统一对物联网终端、数据、应用和服务，以及第三方进行统一管理。具体的应用类型有自动抄表、电梯管理、物流、交通管理等。

整个系统主要由机器到机器（M2M）终端、网络、平台、应用以及运营系统构成。移动通信网络是信息传送的载体，可以采用各种通信方式进行传送，如短信、彩信、IP 等。如果进一步考虑适应不同的网络、考虑接入更大终端数量，以及便于将物联网服务更便利地提供给企业用户，可以分别考虑引入物联网接入网关设备和物联网应用行业网关设备。接入网关设备可以支持物联网终端的汇聚和对不同网络的支持，尤其对网络地址转换（NAT）穿越的支持；行业网关设备将物联网服务或者服务的接口，以行业网关方式提供给企业，有点类似现在的短信网关之类的设备。

（3）协同感知型业务。

在信息汇聚型业务中，物联网的终端只要接受物联网平台管理，执行数据采集、简单处理、上报、接受管理等功能，物联网的终端之间不需要进行通信。

随着物联网的发展，物联网应用应该能够担负起更为重要的任务、更为复杂的业务和服务。这类服务需要物联网终端之间、物联网终端和人之间执行更为复杂的通信，同时这种通信能力在可靠性、时延等方面可能有更高要求，对物联网终端的智能化要求也更为突出，这样才能满足协同处理的要求。

这类应用非常具体的内容，如应用场景、需求、架构、通信协议之类，现在还没有深入研究，但从长远来看，协同感知类业务是物联网发展的趋势。

（4）泛在服务。

泛在服务以无所不在、无所不包、无所不能为基本特征，以实现在任何时间、任何地点、任何人、任何物都能顺畅地通信为目标，是人类通信服务的极致。

物联网的发展是社会前进的必然，拥有着美好的前景，但在发展的道路上，也必将面临多个困难，这些困难有技术上的，更有产业链上的。所以，对于有志于投身物联网行业的程序员而言，机遇与挑战并存，关键还在自己。

12.2.3 你平时读的专业书籍有哪些

“好马配好鞍，好鞍配好马”。一般认为，程序员的个人能力与其阅读的书籍数量、书籍质量存在着巨大的关联关系，所以面试官有时会把求职者阅读的专业书籍作为评价求职者个人水平和能力的重要标准。在他们看来，能够阅读高水平书籍的求职者一般基础知识更加牢固，发展前景更加明朗。所以，如果能够回答出一些比较经典的书籍，无疑对面试的成功会有很大的帮助。

表 12-1 所示为一些经典的计算机类的书籍。

表 12-1 经典计算机类书籍

| 类 别 | 书 籍 |
|------|--------------------------------|
| C | C 专家编程 |
| | C 和指针 |
| | C Primer Plus |
| | The C programming language |
| | C 陷阱与缺陷 |
| C++ | C++程序设计语言（特别版） |
| | C++ Primer （第 3 版）中文版 |
| | C++ Primer （第 4 版）中文版 |
| | C++标准程序库——自修教程与参考手册 |
| | C++语言的设计和演化 |
| | 深度探索 C++对象模型 |
| | Essential C++中文版 |
| | Effective C++中文版 |
| | More Effective C++中文版 |
| | C++编程思想 |
| | C++ Primer Plus |
| | |
| Java | Java 编程语言（第 3 版） |
| | Java 编程思想（第 2 版） |
| | Java 编程思想（第 3 版） |
| | Effective Java 中文版 |
| | Java 2 核心技术 卷 I：基础知识（原书第 7 版） |
| | Java 2 核心技术 卷 II：高级特性（原书第 7 版） |
| VC | 深入浅出 MFC |
| | MFC Windows 程序设计（第 2 版） |
| | Visual C++ 技术内幕（第 4 版） |
| | 深入解析 MFC |
| 算法 | 算法导论 |
| | 计算机程序设计艺术 |
| | 编程之美 |
| 操作系统 | 深入理解计算机系统结构 |
| | Linux 内核设计与实现 |
| | GNU/Linux 编程指南（第 2 版） |
| | Linux 内核完全注释 |
| | Linux 内核分析及编程 |
| 网络编程 | TCP/IP 详解 3 卷本 |
| | UNIX 网络编程 |
| 编译原理 | 编译原理（Alfred V.Aho 等著） |
| | 编译原理基础（刘坚著） |
| 软件工程 | 设计模式——可复用面向对象软件的基础 |
| | 大话设计模式 |
| | 重构——改善既有代码的设计 |

面试官除了询问求职者阅读的计算机类图书外，可能会询问求职者平时关注的技术网站有哪些。国内比较著名的技术网站有 CSDN、51CTO、chinaunix、cnblogs 等，国外比较著名的技术网站有 <http://stackoverflow.com>、<http://www.cplusplus.com/doc/tutorial/>、<http://www.codeproject.com/>等。

数据结构与算法是计算机发展的基石，现代计算机的起源是数学，数学的核心是算法，计算机历史上每一次大的变革都离不开算法的推动。纵然“条条大路通罗马”，但好的算法永远比提高硬件设备管用。

计算机程序的灵魂是数据结构与算法，而随着各种集成开发环境的日益完善，编程已经不再是程序员的专利，短时间内掌握几种开发语言、开发工具已经不再困难。数据结构与算法的好坏则是区分优秀程序员与普通程序员的重要标志之一。鉴于此，绝大多数 IT 企业在面试笔试时都会重点考核求职者对该类知识的掌握程度，并以此作为重要的评价标准。

13.1 数组

数组是程序开发中常见的数据表现形式，由此也非常容易衍生出一些特定的算法设计题目，而在面试笔试中，针对此类算法的考查也非常常见，本节内容几乎囊括了数组相关算法的设计题目。

13.1.1 如何用递归实现数组求和

给定一个含有 n 个元素的整型数组 a ，求 a 中所有元素的和。

如果不要求递归求解，最简单的方法，也是最容易想到的方法只需要进行一次循环，然后求和即可。程序示例如下：

```
#include <stdio.h>

int main()
{
    int a[] = {3,6,8,2,1};
    int i;
    int len = sizeof(a)/sizeof(a[0]);
    int sum = 0;
    for (i=0;i<len;i++)
    {
        sum += a[i];
    }
    printf("%d\n",sum);
    return 0;
}
```

程序输出结果：

20

问题的难点在于如何使用递归上。如果使用递归，则需要考虑如何进行递归执行的开始以及终止条件，首先如果数组元素个数为 0，那么和为 0。同时，如果数组元素个数为 n ，那么先求出前 $n-1$ 个元素之和，再加上 $a[n-1]$ 即可。此时可以完成递归功能。程序代码如下：

```
#include <stdio.h>
```

```

int GetSum(int*a, int n)
{
    return n == 0 ? 0 : GetSum(a, n - 1) + a[n - 1];
}

int main()
{
    int a[] = {3,6,8,2,1};
    int length = sizeof(a)/sizeof(a[0]);
    printf("%d\n",GetSum(a,length));
    return 0;
}

```

程序输出结果:

20

13.1.2 如何用一个 for 循环打印出一个二维数组

常规的可以通过两层 for 循环嵌套来进行二维数组的输出, 设二维数组 `array[MAXX][MAXY]`, 其中 `MAXX` 表示的是二维数组的行数, `MAXY` 表示的是二维数组的列数, 程序代码如下:

```

#include <stdio.h>
#define MAXX 2
#define MAXY 3

void printArray()
{
    int array[MAXX][MAXY]={1,2,3,4,5,6};
    for(int i=0;i<MAXX;i++)
        for(int j=0;j<MAXY;j++)
        {
            printf("%d\n", array[i][j]);
        }
}

int main()
{
    printArray();
    return 0;
}

```

而题目要求是只使用一次 for 循环, 此时就需要明白二维数组在内存中是按照行存储的还是列存储的了 (默认情况是行存储的), 所以可以将数组 `array` 看成一个一维数组, `i` 标识该数组在一维数组中的位置, 则 `array` 在二维数组中的行号和列号分别为 `[i/MAXY]`, `[i%MAXY]`。具体实现代码如下:

```

#include <stdio.h>

#define MAXX 2
#define MAXY 3

void printArray()
{
    int array[MAXX][MAXY]={1,2,3,4,5,6};
    for(int i=0;i<MAXX*MAXY;i++)
    {

```

```

        printf("%d\n", array[i/MAXY][i%MAXY]);
    }
}

int main()
{
    printArray();
    return 0;
}

```

再例如，对于一个三维数组而言，也可以采用类似的方法实现。程序示例代码如下：

```

#include <stdio.h>

int main()
{
    int a[2][2][3]={{1,6,3},{5,4,15}},{3,5,33},{23,12,7}};
    for(int i=0;i<12;i++)
        printf("%d ",a[i/6][(i/3)%2][i%3]);
    printf("\n");
    return 0;
}

```

程序输出结果：

```
1 6 3 5 4 15 3 5 33 23 12 7
```

上例中，需要考虑的是，数组中每一维数字的取值顺序问题。由于该数组为多维数组，第一维，前6次循环都取0，后6次取1，于是 $i/6$ 可以满足要求；第二维，前3次为0，再3次为1，再3次为0，再3次为1，用量化的思想， $i/3$ 把12个数字分为4组，每组3个，量化为0、1、2、3，为了要得到0、1、0、1，这里就需要对 $(0、1、2、3) \% 2 = (0、1、0、1)$ ，于是 $(i/3) \% 2$ ；最后一维需要的是(0、1、2；0、1、2；0、1、2；0、1、2；)即 $i \% 3$ 。

13.1.3 在顺序表中插入和删除一个结点平均移动多少个结点

在等概率情况下，顺序表中插入一个结点需平均移动 $n/2$ 个结点。删除一个结点需平均移动 $(n-1)/2$ 个结点。具体的移动次数取决于顺序表的长度 n 以及需插入或删除的位置 i 。 i 越接近 n 则所需移动的结点数越少。

13.1.4 如何用递归算法判断一个数组是否是递增

判断一个数组中的元素是否递增，最容易想到的解决办法就是遍历数组，然后判断相邻的两个数组元素的大小是否满足下标小的元素其值也越小。如果不满足，则不是递增数组。

本题要求使用递归的算法，设数组为 a ，则递归数组满足以下条件。

- (1) 如果数组长度为1，则该数组为递增，返回 true。
- (2) 若果数组长度为 n ($n \geq 2$)，则先比较最后两个元素是否递增，如果最后两个元素递增，则再递归比较除去最后一个元素的前 $n-1$ 个元素是否递增。

具体实现代码如下所示：

```

#include<stdio.h>

bool isIncrease(int a[], int n)
{
    if( n == 1 )
        return true;
    return ( a[n-1] >= a[n-2] ) && isIncrease( a,n-1);
}

```

```

int main()
{
    int array[]={1,2,3,3,4,5};
    int len = sizeof(array)/sizeof(array[0]);
    if (isIncrease(array,len))
        printf("数组{1,2,3,3,4,5}是递增数组\n");
    else
        printf("数组{1,2,3,3,4,5}不是递增数组\n");
    return 0;
}

```

程序输出结果:

1,2,3,3,4,5 是递增数组

13.1.5 如何分别使用递归与非递归实现二分查找算法

二分查找法也称为折半查找法，它的思想是每次都与序列的中间元素进行比较。二分查找的一个前提条件是数组是有序的，假设数组 `array` 为递增序列，`findData` 为要查找的数，`n` 为数组长度，首先将 `n` 个元素分成个数大致相同的两半，取 `array[n/2]` 与将要查找的值 `findData` 进行比较，如果 `findData` 等于 `array[n/2]`，则找到 `findData`，算法终止；如果 `findData < array[n/2]`，则只要在数组 `array` 的左半部分继续搜索 `findData`；如果 `findData > array[n/2]`，则只需要在数组 `array` 的右半部分继续搜索即可。

二分查找可以使用递归和非递归的方法来解决，以下是代码示例。

```
#include<stdio.h>
```

```
//非递归算法，如果存在返回数组位置，不存在则返回-1
```

```
int BinarySearch(int array[],int len,int findData)
```

```

{
    if(array==NULL||len<=0)
        return -1;
    int start=0;
    int end=len-1;
    while(start<=end)
    {
        int mid=start+(end-start)/2;
        if(array[mid]==findData)
            return mid;
        else if(findData<array[mid])
            end=mid-1;
        else
            start=mid+1;
    }
    return -1;
}

```

```
//递归算法
```

```
int BinarySearchRecursion(int array[],int findData,int start,int end)
```

```

{
    if(start>end)
        return -1;
    int mid=start+(end-start)/2;
    if(array[mid]==findData)
        return mid;
    else if(findData<array[mid])

```

```

        return BinarySearchRecursion(array,findData,start,mid-1);
    else
        return BinarySearchRecursion(array,findData,mid+1,end);
    }

int BinarySearchRecursion(int array[],int len,int findData)
{
    if(array==NULL||len<=0)
        return -1;
    return BinarySearchRecursion(array,findData,0,len-1);
}

int main()
{
    int array[]={1,2,3,4,5,6,7,8};
    int len=sizeof(array)/sizeof(int);
    int index=BinarySearch(array,len,4);
    int index2=BinarySearchRecursion(array,len,9);
    printf("%d\n%d\n",index,index2);
    return 0;
}

```

程序输出结果:

3
-1

需要注意的是，二分查找算法的时间复杂度为 $O(\log n)$ ，最坏情况下的时间复杂度为 $O(n)$ 。

13.1.6 如何在排序数组中，找出给定数字出现的次数

如何在排序数组中，找出给定数字出现的次数？例如，数组[1, 2, 2, 2, 3] 中 2 的出现次数是 3 次。

该问题的解决需要在二分查找法的基础上进行改进。设数组 `array` 为递增序列，需要查找的元素为 `findData`，为了求解给定数字出现的次数，可以分别寻找 `findData` 在 `array` 中最先出现的位置和最后出现的位置，通过两者的算术运算即可获得该数字的出现次数。编码的时候，用一个变量 `last` 来存储本次查找到的位置，然后根据情况变换查找方向，就可以分别确定最先出现的位置的下标 `left` 和最后出现的位置的下标 `right` 的值。

具体实现代码如下所示：

```

#include <stdio.h>

//isLeft 标记的值是否在左边
int BinarySearch(int* a, int length, int num, bool isLeft)
{
    int left = 0, right = length - 1;
    int last = 0;
    while (left <= right)
    {
        int mid = (left + right) / 2;
        if (a[mid] < num)
        {
            left = mid + 1;
        }
        else if (a[mid] > num)
        {

```



```

        right = mid - 1;
    }
    else
    {
        last = mid;
        if (isLeft)
        {
            right = mid - 1;
        }
        else
        {
            left = mid + 1;
        }
    }
}
return last > 0 ? last : -1;
}

int main()
{
    int array[]={0,1,2,3,3,3,3,3,3,3,4,5,6,7,13,19};
    int Lower = BinarySearch(array,sizeof(array)/sizeof(array[0]),3,true);
    int Upper = BinarySearch(array,sizeof(array)/sizeof(array[0]),3,false);
    int count = Upper-Lower+1;
    printf("%d\n",count);
    return 0;
}

```

程序输出结果:

8

13.1.7 如何计算两个有序整型数组的交集

例如，两个含有 n 个元素的有序（非降序）整型数组 a 和 b （数组 a 和 b 中都没有重复元素），求出其共同元素。

$a = 0, 1, 2, 3, 4$

$b = 1, 3, 5, 7, 9$

那么它们的交集为 $\{1, 3\}$ 。

计算数组交集可以采用很多种方法，但数组的相对大小一般会影响算法的效率，所以需要根据两个数组的相对大小来确定采用的方法。

(1) 对于两个数组长度相当的情况，一般可以采取以下 3 种方法。

方法一：采用二路归并来遍历两个数组。

设两个数组分别为 $array1[n1]$ 和 $array2[n2]$ ，分别以 i 、 j 从头开始遍历两个数组。在遍历过程中，如果当前遍历位置的 $array1[i]$ 与 $array2[j]$ 相等，则此数为两个数组的交集，记录下来，并继续向后遍历 $array1$ 和 $array2$ 。如果 $array1[i]$ 大于 $array2[j]$ ，则需继续向后遍历 $array2$ 。如果 $array1[i]$ 小于 $array2[j]$ ，则需要继续向后遍历 $array1$ ，直到有一个数组结束遍历即停止。具体代码如下所示：

```

int mixed(int array1[],int n1,int array2[],int n2,int* mixed)
{
    int i=0,j=0,k=0;
    while(i<n1&&j<n2)
    {

```

```

        if(array1[i]==array2[j])
        {
            mixed[k++]=array1[i];
            i++;
            j++;
        }
        else if(array1[i]>array2[j])
        {
            j++;
        }
        else if(array1[i]<array2[j])
        {
            i++;
        }
    }
    return k;
}

```

方法二：顺序遍历两个数组，将数组元素存放到哈希表中，同时对统计的数组元素进行计数。如果为 2，则为两者的交集元素。

方法三：遍历两个数组中任意一个数组，将遍历得到的元素存放到哈希表，然后遍历另外一个数组，同时对建立的哈希表进行查询，如果存在，则为交集元素。

(2) 对于两个数组长度相差悬殊的情况，如数组 a 的长度远远大于数组 b 的长度，则可以采用下面几种方法。

方法一：依次遍历长度小的数组，将遍历的得到的数组元素在长数组中进行二分查找。具体而言，设两个指向两个数组末尾元素的指针，取较小的那个数在另一个数组中二分查找，找到，则存在一个交集，并且将该目标数组的指针指向该位置前一个位置。如果没有找到，同样可以找到一个位置，使得目标数组中在该位置后的数肯定不在另一个数组中存在，直接移动该目标数组的指针指向该位置的前一个位置，再循环找，直到一个数组为空为止。因为两个数组中都可能出现重复的数，因此二分查找时，当找到一个相同的数 x 时，其下标为 i，那么下一个二分查找的下界变为 i+1，避免 x 重复使用。

方法二：采用与方法一类似的方法，但是每次查找在前一次查找的基础上进行，这样可以大大缩小查找表的长度。

方法三：采用与方法二类似的方法，但是遍历长度小的数组的方式有所不同，从数组头部和尾部同时开始遍历，这样可以进一步缩小查找表的长度。

13.1.8 如何找出数组中重复次数最多的数

例如，数组 {1,1,2,2,4,4,4,4,5,5,6,6,6}，元素 1 出现的次数为两次，元素 2 出现的次数为两次，元素 4 出现的次数为 4 次，元素 5 出现的次数为两次，元素 6 出现的次数为 3 次，问题就是要找出出现重复次数最多的数，所以输出应该为元素 4。可以采取如下两种方法来计算数组中重复次数最多的数。

方法一：以空间换时间，可以定义一个数组 int count[MAX]，并将其数组元素都初始化为 0，然后执行 for(int i = 0; i < 100; i++) count[A[i]]++; 操作，在 count 中找最大的数，即为重复次数最多的数。

程序示例如下：

```
#include <stdio.h>
```

```

int GetMaxNum(int* arr, int len, int& num)
{
    int index = arr[0];
    int i;
    for(i=0; i<len; i++)
    {
        if (arr[i]>index)
        {
            index = arr[i];
            num = i;
        }
    }
    return index;
}

int main()
{
    int array[]={1,1,2,2,4,4,4,4,5,5,6,6};
    int length = sizeof(array)/sizeof(array[0]);
    int i;
    int num = 0;
    int* count = new int[GetMaxNum(array,length,num)];
    for(i=0;i<length;i++)
        count[i] = 0;
    for(i=0;i <length;i++)
        count[array[i]]++;
    printf("最大的数出现的次数是: %d\n",GetMaxNum(count,GetMaxNum(array,length,num),num));
    printf("最大的数是: %d\n",num);
    return 0;
}

```

程序输出结果:

最大的数出现的次数是: 4
最大的数是: 4

上例是一种典型的空间换时间算法。一般情况下,除非内存空间足够大,否则一般不采用这种方法。

方法二: 使用 map 映射表,通过引入 map 表(map 是 STL 的一个关联容器,它提供一对一的数据处理能力,其中第一个为关键字,每个关键字只能在 map 中出现一次,第二个称为该关键字的值)来记录每一个元素出现的次数,然后判断次数大小,进而找出重复次数最多的元素。

程序示例如下:

```

#include <iostream>
#include <map>
using namespace std;

bool findMostFrequentInArray(int *a, int size, int &val)
{
    if(size == 0)
        return false;
    map<int, int> m;
    for (int i = 0; i < size; i++)
    {
        if(++m[a[i]] >= m[val])
            val = a[i];
    }
}

```

```

        return true;
    }

    int main()
    {
        int a[]={1, 2, 3, 4, 4, 4, 5, 5, 5, 5, 6};
        int val = 0;
        if(findMostFrequentInArray(a, 11, val))
            cout << val << endl;
        int b[]={1, 5, 4, 3, 4, 4, 5, 4, 5, 5, 6};
        if (findMostFrequentInArray(b, 11, val))
            cout << val << endl;
        int c[]={1, 5, 4, 3, 4, 4, 5, 4, 5, 5, 6, 6, 6, 6, 6};
        if (findMostFrequentInArray(c, 15, val))
            cout << val << endl;
        return 0;
    }

```

程序输出结果:

```

5
5
6

```

13.1.9 如何在 $O(n)$ 的时间复杂度内找出数组中出现次数超过了一半的数

如果本题对时间复杂度没有要求,可以采用很多方法。第一种方法是建立一个二维数组,一维存储数组中的数据,二维存这个数出现的次数,出现次数最多的那个数就是要找的那个数。由于某个数出现的次数超过数组长度的一半,所以二维数组的长度只需要这个数组的一半即可,但这种方法的时间复杂度和空间复杂度都比较大。

第二种方法是先对数组排序,然后取中间元素即可,因为如果某个元素的个数超过一半,那么数组排序后该元素必定占据数组的中间位置。如果出现最多的那个数是最小的,那么 $1 \sim (n+1)/2$ 都是那个数;如果出现最多的那个数是最大的,那么 $(n-1)/2 \sim n$ 都是那个数;如果不是最小也不是最大,当这个数由最小慢慢变成最大的数时,会发现中间的那个数的值是不变的,所以中间那个数就是要找的那个数。时间复杂度就是排序用的时间,即最快的排序算法的时间复杂度 $O(n \log n)$ 。

但由于本题对时间复杂度有要求,以上方法显然都达不到要求,不可取,需要采取非常规方法,利用一些其他技巧来实现,于是想到了以下几种方法。

方法一:每次取出两个不同的数,剩下的数字中重复出现的数字肯定比其他数字多,将规模缩小化。如果每次删除两个不同的数(不管包括不包括最高频数),那么在剩余的数字里,原最高频数出现的频率一样超过了 50%,不断重复这个过程,最后剩下的将全是同样的数字,即最高频数。此算法避免了排序,时间复杂度只有 $O(n)$ 。

程序示例如下:

```

#include <stdio.h>

int FindMostApperse(int* num,int len)
{
    int candidate = 0;
    int count = 0;
    for (int i = 0; i < len; i++)
    {
        if (count == 0)

```

```

        {
            candidate = num[i];
            count = 1;
        }
        else
        {
            if (candidate == num[i])
                count++;
            else
                count--;
        }
    }
    return candidate;
}

int main()
{
    int arr[] = {2,1,1,2,3,1,1,1};
    int len = sizeof(arr)/sizeof(arr[0]);
    printf("%d\n",FindMostApperse(arr,len));
    return 0;
}

```

程序输出结果:

1

方法二: Hash 法。首先创建一个 hash_map, 其中 key 为数组元素值, value 为此数出现的次数。遍历一遍数组, 用 hash_map 统计每个数出现的次数, 并用两个值存储目前出现次数最多的数和对应出现的次数, 此时的时间复杂度为 $O(n)$, 空间复杂度为 $O(n)$, 满足题目的要求。

方法三: 使用两个变量 A 和 B, 其中变量 A 存储某个数组中的数, 变量 B 用来计数。开始时将变量 B 初始化为 0, 遍历数组:

如果当前数与 A 不同, 则需要分两种情况进行讨论:

(1) 如果 B 等于 0, 则令 A 等于当前数, 令 B 等于 1。

(2) 如果 B 大于 0, 则令 $B=B-1$ 。

如果当前数与 A 相同, 则令 $B=B+1$ 。遍历结束时, A 中存储的数就是所要找的数。这个算法的时间复杂度是 $O(n)$, 空间复杂度为 $O(1)$ 。

具体代码如下所示:

```

#include <stdio.h>

int main()
{
    int i,A,B;
    int a[10]={1,2,3,1,2,1,1,6,1,1};
    A=a[5];
    B=0;
    for(i=0;i<10;i++)
    {
        if(B==0)
        {
            A=a[i];
            B=1;
        }
        else if(A==a[i])
        {

```



```

        B++;
    }
    else if(A!=a[i])
    {
        B--;
    }
}
printf("%d\n",A);
return 0;
}

```

程序输出结果:

1

13.1.10 如何找出数组中唯一的重复元素

数组 $a[N]$, 1 至 $N-1$ 这 $N-1$ 个数存放在 $a[N]$ 中, 其中某个数重复一次, 写一个函数, 找出被重复的数字。要求每个数组元素只能访问一次, 不用辅助存储空间。

由于题目要求每个数组元素只能访问一次, 不用辅助存储空间, 可以从原理上入手, 采用数学求和法, 因为只有一个数字重复一次, 而数又是连续的, 根据累加和原理, 对数组的所有项求和, 然后减去 1 至 $N-1$ 的和, 即为所求的重复数。程序代码如下:

```

#include <stdio.h>

void xor_findDup(int * a,int N)
{
    int tmp1 = 0;
    int tmp2 = 0;
    for (int i=0; i<N-1; ++i)
    {
        tmp1+=(i+1);
        tmp2+=a[i];
    }
    tmp2+=a[N-1];
    int result=tmp2-tmp1;
    printf("%d\n",result);
}

int main()
{
    int a[] = {1,2,1,3,4};
    xor_findDup(a,5);
    return 0;
}

```

程序输出结果:

1

如果题目没有要求每个数组元素只能访问一次, 不用辅助存储空间, 还可以用异或法和位图法来求解。

(1) 异或法。

根据异或法的计算方式, 每两个相异的数执行异或运算之后, 结果为 1; 每两个相同的数异或之后, 结果为 0, 所以数组 $a[N]$ 中的 N 个数异或结果与 1 至 $N-1$ 异或的结果再做异或, 得到的值即为所求。

设重复数为 A , 其余 $N-2$ 个数异或结果为 B , N 个数异或结果为 A^A^B , 1 至 $N-1$ 异或结果为 A^B , 由于异或满足交换律和结合律, 且 $X^X=0$, $0^X=X$, 则有 $(A^B)^{(A^A^B)}=$

$A \oplus B \oplus B = A$ 。

程序代码如下：

```
#include <stdio.h>

void xor_findDup(int * a,int N)
{
    int i;
    int result=0;
    for(i=0;i<N;i++)
    {
        result ^= a[i];
    }
    for (i=1;i<N;i++)
    {
        result ^= i;
    }
    printf("%d\n",result);
}

int main()
{
    int a[] = {1,2,1,3,4};
    xor_findDup(a,5);
    return 0;
}
```

程序输出结果：

1

(2) 位图法。

位图法的原理是首先申请一个长度为 $N-1$ 且均为 '0' 组成的字符串，然后从头开始遍历数组 $a[N]$ ，取每个数组元素 $a[i]$ 的值，将其对应的字符串中的相应位置置 1，如果已经置过 1，那么该数就是重复的数。由于采用的是位图法，所以空间复杂度比较大，为 $O(N)$ 。

程序示例如下：

```
#include <stdio.h>
#define sum(x) (x*(x+1)/2)

void xor_findDup(int * arr,int NUM)
{
    int *arrayflag = (int *)malloc(NUM*sizeof(int));
    int i=1;
    while(i<NUM)
    {
        arrayflag[i] = false;
        i++;
    }
    for( i=0; i<NUM; i++)
    {
        if(arrayflag[arr[i]] == false)
            arrayflag[arr[i]] = true;
        else
        {
            printf("%d\n",arr[i]);
            return ;
        }
    }
}
```

```

    }

    int main()
    {
        int a[] = {1,2,1,3,4};
        xor_findDup(a,5);
        return 0;
    }

```

程序输出结果：

1

此题可以进行一个变形：取值为 $[1, n-1]$ 含 n 个元素的整数数组，至少存在一个重复数，即可能存在多个重复数， $O(n)$ 时间内找出其中任意一个重复数。例如，`array[]={1,2,2,4,5,4}`，则 2 和 4 均是重复元素。

方案一，位图法。使用大小为 N 位图，记录每个元素是否出现过，一旦遇到一个已经出现过的元素，则直接输出。时间复杂度是 $O(N)$ ，空间复杂度为 $O(N)$ 。

方案二，数组排序法。首先对数组进行计数排序，然后顺次扫描整个数组，直到遇到一个已出现的元素，直接将之输出。时间复杂度为 $O(N)$ ，空间复杂度为 $O(N)$ 。

以上提出的两种方案都需要额外的存储空间，能不能不使用额外存储空间呢？答案是肯定的。于是想到了方案三的 Hash 法。数组元素如果是有符号的 `int` 型，则本方法可行。将数组元素值作为索引，对于元素 `array[i]`，如果 `array[array[i]]` 大于 0，则设置 `array[array[i]] = -array[array[i]]`；如果 `array[array[i]]` 小于 0，则 `array[array[i]]` 是一个重复数，直接输出，最后还原 `array` 中各个被修改的元素。

具体算法如下：

```

int FindInteger(int array[], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        if(array[i] > 0)
        {
            if(array[array[i]] > 0)
            {
                array[array[i]] = -array[array[i]];
            }
            else
            {
                return -array[array[i]];
            }
        }
        else
        {
            if(array[-array[i]] > 0)
            {
                array[array[i]] = -array[array[i]];
            }
            else
            {
                return -array[-array[i]];
            }
        }
    }
}

```

方法四是一种非常诡异的算法，就是采用类似于单链表是否存在环的问题。“判断单链表是否存在环”是一个非常经典的问题，同时单链表可以采用数组实现，此时每个元素值作为 next 指针指向下一个元素。本题可以转化为“已知一个单链表中存在环，找出环的入口点”这种想法。具体思路如下：将 `array[i]` 看做第 `i` 个元素的索引，即 `array[i] → array[array[i]] → array[array[array[i]]] → array[array[array[array[i]]]] → ...` 最终形成一个单链表，由于数组 `a` 中存在重复元素，则一定存在一个环，且环的入口元素即为重复元素。

该题的关键在于，数组 `array` 的大小是 `n`，而元素的范围是 `[1,n-1]`，所以 `array[0]` 不会指向自己，进而不会陷入错误的自循环。如果元素的范围中包含 0，则该题不可直接采用该方法。程序示例代码如下：

```
#include <stdio.h>

int FindInteger(int array[], int n)
{
    int x, y;
    x = y = 0;
    do
    {
        x = array[array[x]]; //x 一次走两步
        y = array[y]; //y 一次走一步
    } while(x != y); //找到环中的一个点
    x = 0;
    do
    {
        x = array[x];
        y = array[y];
    } while(x != y); //找到入口点
    return x;
}

int main()
{
    int array[] = {1,2,2,4,5,4};
    int length = sizeof(array)/sizeof(array[0]);
    printf("%d\n",FindInteger(array,length));
    return 0;
}
```

程序输出结果：

2

13.1.11 如何判断一个数组中的数值是否连续相邻

一个整数数列，元素取值可能是 `0~65535` 中的任意一个数，相同数值不会重复出现；0 是例外，可以反复出现。设计一个算法，当从该数列中随意选取 5 个数值时，判断这 5 个数值是否连续相邻。需要注意以下 4 点：

- (1) 5 个数值允许是乱序的，如 8 7 5 0 6。
- (2) 0 可以通配任意数值，如 8 7 5 0 6 中的 0 可以通配成 9 或者 4。
- (3) 0 可以多次出现。
- (4) 全 0 算连续，只有一个非 0 算连续。

如果没有 0 的存在，要组成连续的数列，最大值和最小值的差距必须是 4，存在 0 的情况下，只要最大值和最小值的差距小于 4 就可以了。所以找出数列中非 0 的最大值和非 0 的最小

值, 时间复杂度为 $O(n)$ 。如果非 0 最大-非 0 最小+1 \leq 5 (即非 0 最大-非 0 最小 \leq 4) 则这 5 个数值连续相邻。否则, 不连续相邻。因此, 总体复杂度为 $O(n)$ 。

程序示例代码如下:

```
#include<stdio.h>

bool IsContinuous(int* a,int n)
{
    int min = -1,max = -1;
    for (int i = 0;i < n;i++)
    {
        if (a[i] != 0)
        {
            if (min > a[i] || -1 == min)
                min = a[i];
            if (max < a[i] || -1 == max)
                max = a[i];
        }
    }
    if (max - min > n - 1)
        return false;
    else
        return true;
}

int main()
{
    int array[]={8,7,5,0,6};
    int len = sizeof(array)/sizeof(array[0]);
    if (IsContinuous(array,len))
        printf("数组 {8,7,5,0,6} 连续相邻\n");
    else
        printf("数组 {8,7,5,0,6} 不连续相邻\n");
    return 0;
}
```

程序输出结果:

8,7,5,0,6 连续相邻

13.1.12 如何找出数组中出现奇数次的元素

给定一个含有 n 个元素的整型数组 `array`, 其中只有一个元素出现奇数次, 找出这个元素。

因为对于任意一个数 k , 有 $k \wedge k = 0$, $k \wedge 0 = k$, 所以将 `array` 中所有元素进行异或, 那么个数为偶数的元素异或后都变成了 0, 只留下了个数为奇数的那个元素。

程序示例如下:

```
#include<stdio.h>

int FindElementWithOddCount(int*a, int n)
{
    int r = a[0];
    for (int i = 1; i < n; ++i)
    {
        r ^= a[i];
    }
    return r;
}
```



```

    }

    int main()
    {
        int array[]={1,2,2,3,3,4,1};
        int len = sizeof(array)/sizeof(array[0]);
        printf("%d\n",FindElementWithOddCount(array,len));
        return 0;
    }

```

程序输出结果:

4

引申: 由 n 个元素组成的数组, $n-2$ 个数出现了偶数次, 两个数出现了奇数次 (这两个数不相等), 如何用 $O(1)$ 的空间复杂度, 找出这两个数?

假设这两个数分为 a 、 b , 将数组中所有元素异或之后结果为 x , 因为 $a \neq b$, 所以 $x = a \oplus b$, 且 $x \neq 0$, 判断 x 中位为 1 的位数, 只需要知道某一个位为 1 的位数 k , 如 00101100, k 可以取 2 或者 3, 或者 5, 然后将 x 与数组中第 k 位为 1 的数进行异或, 异或结果就是 a 或 b 中的一个, 然后用 x 异或, 就可以求出另外一个。

因为 x 中第 k 位为 1 表示 a 或 b 中有一个数的第 k 位也为 1, 假设为 a , 将 x 与数组中第 k 位为 1 的数进行异或时, 也即将 x 与 a 以及其他第 k 位为 1 的出现过偶数次的数进行异或, 化简即为 x 与 a 异或, 最终结果即为 b 。

程序示例如下:

```

#include <stdio.h>

void FindElement(int a[],int length)
{
    int s=0;
    int i;
    int k=0;
    for(i=0;i<length;i++)
    {
        s=s^a[i];
    }
    int s1=s;
    int s2=s;
    while(!(s1&1))
    {
        s1=s1>>1;
        k++;
    }
    for(i=0;i<length;i++)
    {
        if((a[i]>>k)&1)
            s=s^a[i];
    }
    printf("%d %d\n",s,s^s2);
}

int main()
{
    int array[]={1,2,2,3,3,4,1,5};
    int len = sizeof(array)/sizeof(array[0]);
    FindElement(array,len);
    return 0;
}

```

```

    }
    程序输出结果:
    4 5

```

13.1.13 如何找出数列中符合条件的数对的个数

一个整数数组，元素取值可能是 $1 \sim N$ (N 是一个较大的正整数) 中的任意一个数，相同数值不会重复出现。设计一个算法，找出数列中符合条件的数对的个数，满足数对中两数的和等于 $N+1$ 。

方法一：蛮力法。这是最简单的方法，枚举出数组中所有可能的数对，看其和是否为 $N+1$ ，如果是，则输出。但这种方法一般效率不高。

方法二：先对数组进行排序，然后使用二分查找方法，用两个指示器 (`front` 和 `back`) 分别指向第一个和最后一个元素，然后从两端同时向中间遍历，直到两个指针交叉。

(1) 如果 $A[\text{front}] + A[\text{back}] > N+1$ ，则 `back--`。

(2) 如果 $A[\text{front}] + A[\text{back}] = N+1$ ，则计数器加 1，`back--`，同时 `front++`。

(3) 如果 $A[\text{front}] + A[\text{back}] < N+1$ ，则 `front++`。

重复上述步骤， $O(n)$ 时间就可以找到所有数对，因此总体复杂度为 $O(n \log n)$ 。

程序代码如下：

```

#include<stdio.h>

void FixedSum(int* a, int n, int d)
{
    for (int i = 0, j = n - 1; i < n && j >= 0 && i < j;)
    {
        if (a[i] + a[j] < d)
            ++i;
        else if (a[i] + a[j] == d)
        {
            printf("%d,%d\n", a[i], a[j]);
            ++i;
            --j;
        }
        else
            --j;
    }
}

int main()
{
    int array[] = {1, 2, 3, 4, 5};
    int len = sizeof(array) / sizeof(array[0]);
    FixedSum(array, len, 6);
    return 0;
}

```

程序输出结果：

```

1,5
2,4

```

方法三：用计数排序，将 $1 \sim N$ 个数放在一块很大的空间里面，比如 1 放在 1 号位， N 放在 n 号位置， $O(n)$ 的时间复杂度，然后取值，也是 $O(n)$ 的复杂度。因此，总体复杂度为 $O(n)$ 。

引申：如果是任意数组而不是本题的有规律数组，如何求解数组对？即给定一个任意整

数数组 `array[n]`，寻找数组中和值为 `SUM` 的数对。

最容易想到的就是两重循环迭代，对数组中任意两个数进行求和，看其值是否等于 `SUM`。由于需要两重迭代，所以时间复杂度为 $O(n*n)$

例如：

```
for (int i = 0; i < n; ++i)
{
    for (int j = i+1; j < n; ++j)
    {
        ...
    }
}
```

上述方法时间复杂度太高，其实可以参照题目的方法二，先将数组排序后（一般最快的排序算法时间复杂度为 $O(n\log n)$ ），然后设两个指针指向数组两端，判断两个指针对应元素之和是否为 `SUM`，如果等于 `SUM`，则找到了，继续查找；如果小于 `SUM`，那么首指针递增；如果大于 `SUM`，尾指针递减，直到两个指针相遇时，如果还是没有和为 `SUM` 的元素对出现，那么返回 `false`。

除了上述方法外，还可以参照上例中的方法三，将数组存储到 `hash` 表中，对每个数 `m`，在 `hash` 表中寻找 `SUM-m`，此时时间复杂度为 $O(n)$ 。需要注意的是，如果数组空间很大，超过了内存的容量，那么可以按照 `hash(max(m, SUM-m))%g`，将数据分到 `g` 个小的组中，然后对每个小组进行单独处理，此时时间复杂度还是 $O(n)$ 。

引申：已知大小分别为 `m`、`n` 的两个无序数组 `A`、`B` 和一个数常数 `c`，求满足 `A[i] + B[j] = c` 的所有 `A[i]` 和 `B[j]`。

方法一：枚举法。该方法是最容易、也是最简单的方法，枚举出数组 `A` 和数组 `B` 中所有的元素对，判断其和是否为 `c`，如果是，则输出。

方法二：排序+二分查找法。首先，对两个数组中较大数组（不妨设为 `A`）排序；然后，对于 `B` 中每个元素 `B[i]` 在 `A` 中二分查找 `c-B[i]`，如果找到，直接输出。此方法的时间复杂度为 $O(m\log m + n\log m)$ 。

方法三：排序+线性扫描法。该方法是方案二的进一步加强，需要对两个数组排序。首先，对 `A` 和 `B` 进行排序；然后用指针 `p` 从头扫描 `A`，用指针 `q` 从尾扫描 `B`，如果 `A[p] + B[q] = c`，则输出 `A[p]` 和 `B[q]`，且 `p++`、`q--`；如果 `A[p] + B[q] > c`，则 `q--`；否则 `p++`。时间复杂度为 $O(m\log m + n\log n)$ 。

算法如下：

```
void print_pairs_with_sum(int A[], int B[], int m, int n, int sum)
{
    sort(A, A + m);
    sort(B, B + n);
    int p, q;
    p = 0, q = n-1;
    while(p < m && q >= 0)
    {
        if(A[p] + B[q] == sum)
        {
            cout << "(" << A[p] << "," << B[q] << ")" << endl;
            p++, q--;
        }
        else if(A[p] + B[q] > sum)
        {
            q--;
        }
        else
        {
            p++;
        }
    }
}
```

```

        q--;
    }
    else
    {
        p++;
    }
}
}

```

方法四：Hash 法。首先，将两个数组中较小的数组（不妨设为 A）保存到 HashTable 中，然后，对于 B 中每个元素 B[i]，也采用相同的 hash 算法在 HashTable 中查找 c-B[i] 是否存在，如果存在，则输出。时间复杂度为 $O(m+n)$ ，空间复杂度为 $O(\min\{m,n\})$ 。

算法如下：

```

void print_pairs_with_sum2(int A[], int B[], int m, int n, int sum)
{
    map<int, bool> hash_table;
    int *psmaller = A;
    int *pbigger = B;
    int nsmaller = (m >= n) ? n : m;
    int nbigger = (m >= n) ? m : n;
    if(m > n)
    {
        psmaller = B;
        pbigger = A;
    }
    for(int i = 0; i < nsmaller; i++)
    {
        hash_table.insert(pair<int, bool>(psmaller[i], true));
    }
    for(int i = 0; i < nbigger; i++)
    {
        if(hash_table.find(sum - pbigger[i]) != hash_table.end())
        {
            cout << "(" << pbigger[i] << "," << sum - pbigger[i] << ")" << endl;
        }
    }
}

```

13.1.14 如何寻找出数列中缺失的数

给一个由 $n-1$ 个数组成的未排序的序列，其元素都是 $1 \sim n$ 中的不同的整数。如何寻找序列中缺失的整数？请写出一个线性时间算法。

可以通过累加求和。首先将该 $n-1$ 个整数相加，得到 sum，然后用 $(1+n)n/2$ 减去 sum，得到的差即为缺失的整数。因为 $1 \sim n$ 一共 n 个数， n 个数的和为 $(1+n)n/2$ ，而未排序数列的和为 sum，多余的这个数即为缺失的数目。

程序示例如下：

```

#include <stdio.h>
#define MAX 5

int main()
{
    int array[MAX] = {3,2,1,6,4};
    int i;
    int sum = 0;

```

```

    int temp;
    for(i=0;i<MAX;i++)
        sum+=i;
    temp = (MAX+1)*(MAX)/2-sum;
    printf("%d\n",temp);
    return 0;
}

```

程序输出结果:

5

13.1.15 如何判定数组是否存在重复元素

假设数组 a 有 n 个元素，元素取值范围是 $1 \sim n$ ，如何判定数组是否存在重复元素？

方法一：对数组进行排序（可以效率比较高的排序算法，如快速排序、堆排序等），然后比较相邻的元素是否相同。时间复杂度为 $O(n \log n)$ ，空间复杂度为 $O(1)$ 。

程序示例如下：

```

#include <stdio.h>
#include <stdlib.h>

int comp(const void *a, const void *b)
{
    return (*(int *)a - *(int *)b);
}

int isArrayRepeat(int *a, int n)
{
    int i = 0;
    if(!a || n<1)
        return -1;
    qsort(a, n, sizeof(int), comp);
    for(i = 0; i < n-1; i++)
    {
        if(a[i] == a[i+1])
        {
            return 1;
        }
    }
    return 0;
}

int main()
{
    int result = -1;
    int a[10] = {10, 9, 5, 4, 7, 6, 3, 2, 1, 9};
    result = isArrayRepeat(a, 10);
    if (result)
        printf("yes\n");
    else
        printf("no\n");
    return 0;
}

```

程序输出结果:

yes

方法二：使用 bitmap（位图）方法。定义长度为 $N/8$ 的 `char` 数组，每个 bit 表示对应数字

是否出现过。遍历数组，使用 `bitmap` 对数字是否出现进行统计。时间复杂度为 $O(n)$ ，空间复杂度为 $O(n)$ 。

方法三：遍历数组，假设第 i 个位置的数字为 j ，则通过交换将 j 换到下标为 j 的位置上。直到所有数字都出现在自己对应的下标处，或发生了冲突。此时的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

程序示例如下：

```
#include <stdio.h>

int isArrayRepeat(int *a, int n)
{
    int i = 0;
    int j = -1;
    for(i = 0; i < n; i++)
    {
        j = a[i];
        if(i == j)
        {
            continue;
        }
        if(a[i] == a[j])
        {
            return 1;
        }
        a[i] = a[j];
        a[j] = j;
    }
    return 0;
}

int main()
{
    int result = -1;
    int a[10] = {10, 9, 5, 4, 7, 6, 3, 2, 1, 9};
    result = isArrayRepeat(a, 10);
    if (result)
        printf("yes\n");
    else
        printf("no\n");
    return 0;
}
```

程序输出结果：

yes

如果数组中只有一个元素重复，还可以使用累加求和的方式来执行。

13.1.16 如何重新排列数组使得数组左边为奇数，右边为偶数

给定一个存放整数的数组，如何重新排列数组使得数组左边为奇数，右边为偶数？要求：空间复杂度为 $O(1)$ ，时间复杂度为 $O(N)$ 。

类似快速排序的处理。可以用两个指针分别指向数组的头和尾，头指针正向遍历数组，找到第一个偶数，尾指针逆向遍历数组，找到第一个奇数，交换两个指针指向的数字，然后两指针沿着相应的方向继续向前移动，重复上述步骤，直到头指针大于等于尾指针为止。具体实现如下：

```

#include<iostream>
using namespace std;

void Swap(int& a,int& b)
{
    int temp = a;
    a = b;
    b = temp;
}

void ReverseArray(int arr[],int len)
{
    int begin = 0;
    int end = len - 1;
    while(begin < end)
    {
        while(arr[begin]%2 == 1 && end > begin)
        {
            ++begin;
        }
        while(arr[end]%2 == 0 && end > begin)
        {
            --end;
        }
        Swap(arr[begin], arr[end]);
    }
}

int main()
{
    int array[]={1, 23, 2, 34, 21, 45, 26, 22, 41, 66, 74, 91, 17, 64} ;
    int len = sizeof(array)/sizeof(array[0]);
    int i;
    printf("原数组为: ");
    for(i = 0;i<len;i++)
        printf("%d ",array[i]);
    printf("\n");
    ReverseArray(array,len);
    printf("经过变换后的数组为: ");
    for(i = 0;i<len;i++)
        printf("%d ",array[i]);
    printf("\n");
    return 0;
}

```

程序输出结果:

原数组为: 1 23 2 34 21 45 26 22 41 66 74 91 17 64

经过变换后的数组为: 1 23 17 91 21 45 41 22 26 66 74 34 2 64

13.1.17 如何把一个整型数组中重复的数字去掉

方法一，也是最容易想到的，就是遍历数组中的每一个元素，将元素与它前面的已经遍历过的元素进行逐一比较，如果发现与前面的数字有重复，则去掉；如果没有重复，则继续遍历下一个元素。由于每一个元素都要与之前所有的元素进行比较，所以时间复杂度为 $O(n^2)$ 。

方法一中这种最原始的方法在 n 比较小时，效率低下的缺点并不明显，但当数组元素比较多时，效率就会非常低下，于是想到了方法二，将原数组的下标值存在一个辅助数组中（也就

是说辅助数组为 {0, 1, 2, 3...}), 然后根据下标指向的值对辅助数组排序, 对排序后的辅助数组去重 (也是根据下标指向的值)。然后再按下标本身的值对去重后的辅助数组排序。之后顺次读出剩下的各下标指向的值即可。

例如, 原数组为 {1, 2, 0, 2, -1, 999, 3, 999, 88}, 则辅助数组为 {0, 1, 2, 3, 4, 5, 6, 7, 8}, 对辅助数组按下标指向的值排序的结果为 {4, 2, 0, 1, 3, 6, 8, 5, 7}, 对辅助数组按下标指向的值去重的结果为 {4, 2, 0, 1, 6, 8, 5}, 对辅助数组按下标本身排序的结果为 {0, 1, 2, 4, 5, 6, 8}, 最后得到的结果为 {1, 2, 0, -1, 999, 3, 88}。主要的时间在两次排序, 所以时间复杂度为 $O(n\log n)$ 。

方法三: 首先通过快速排序, 时间复杂度为 $O(n\log n)$, 然后对排好序的数组经过一次遍历, 将其重复元素通过交换, 最终达到删除重复元素的目的。以数组 $a[5]=\{1,2,1,2,3\}$ 为例, 经过快速排序后, 数组序列变为 {1,1,2,2,3}, 此时标记两个变量 $k=0, i=1$, 此时 $(a[k]=a[0])=(a[1]=a[i])$, 于是执行 $i++$, i 变为 2, $(a[i]=a[2])!=(a[0]=a[k])$, 则执行 $k++$; k 变为 1, $a[1]=a[2]=2$, 然后执行 $i++$; i 变为 3, 继续执行, $(a[i]=a[3])=(a[1]=a[k])$, 于是执行 $i++$; i 变为 4, $(a[i]=a[4])!=(a[1]=a[k])$, 则执行 $k++$; k 变为 2, $a[2]=a[4]=3$, 执行完毕, 返回 k 的值, 即去除重复数字后的数组长度为 3。所以, 总的时间复杂度为 $O(n\log n)$ 。程序代码示例如下:

```
#include <stdio.h>
#include <stdlib.h>

int int_cmp(const void *a, const void *b)
{
    const int *ia = (const int *)a;
    const int *ib = (const int *)b;
    return *ia - *ib;
}

int unique(int *array, int number)
{
    int k = 0;
    for (int i = 1; i < number; i++)
    {
        if (array[k] != array[i])
        {
            array[k+1] = array[i];
            k++;
        }
    }
    return (k+1);
}

int Unique_QuickSortMethod(int *arr, int elements)
{
    //C 语言自带的排序函数
    qsort(arr, elements, sizeof(int), int_cmp);
    return unique(arr, elements);
}

int main()
{
    int array[5] = {1,2,1,2,3};
    int len = sizeof(array) / sizeof(array[0]);
    int size = Unique_QuickSortMethod(array, len);
    for (int i = 0; i < size; i++)
```

```

        printf("%d ", array[i]);
    printf("\n");
    return 0;
}

```

程序输出结果

1 2 3

13.1.18 如何找出一个数组中第二大的数

如果仅仅只考虑实现，而不考虑时间效率，可以首先通过排序算法，将数组进行排序，然后根据数据下标来访问数组中第二大的数，最快的排序算法一般为快速排序算法，但是其时间复杂度仍然为 $O(n\log n)$ ，根据下标访问需要遍历一遍数组，时间复杂度为 $O(n)$ ，所以总的时间复杂度为 $O(n\log n)$ 。

有没有更好的方法能将时间复杂度降低了？答案是肯定的，可以只通过一遍扫描数组即可找出数组中第二大的数，即通过设置两个变量来进行判断。首先定义一个变量来存储数组的最大数，初始值为数组首元素；另一个变量用来存储数组元素的第二大的数，初始值为最小负整数-32767，然后遍历数组元素。如果数组元素的值比最大数变量的值大，则将第二大变量的值更新为最大数变量的值，最大数变量的值更新为该元素的值；如果数组元素的值比最大数的值小，则判断该数组元素的值是否比第二大数的值大，如果大，则更新第二大数的值为该数组元素的值。

程序代码示例如下：

```

#include <stdio.h>

const int MINNUMBER = -32767;
int FindSecMax( int data[], int count)
{
    int maxnumber = data[0];
    int sec_max = MINNUMBER;
    for ( int i = 1; i < count; i++)
    {
        if ( data[i] > maxnumber )
        {
            sec_max = maxnumber;
            maxnumber = data[i];
        }
        else
        {
            if ( data[i] > sec_max )
                sec_max = data[i];
        }
    }
    return sec_max;
}

int main()
{
    int array[] = {2,5,6,7,7,8,98,3,458,5,6};
    int length = sizeof(array)/sizeof(array[0]);
    printf("%d\n",FindSecMax(array,length));
    return 0;
}

```

程序的输出结果：

13.1.19 如何寻找数组中的最小值和最大值

对于本题，一般有以下 5 种解法：

(1) 问题分解法。

把本题看做是两个独立的问题，每次分别找出最小值和最大值，此时需要遍历两次数组，比较次数为 $2N$ 次。

(2) 取单元素法。

维持两个变量 `min` 和 `max`，`min` 标记最小值，`max` 标记最大值，每次取出一个元素，先与已找到的最小值比较，再与已找到的最大值比较。此种方法只需要遍历一次数组即可。

(3) 取双元素法。

维持两个变量 `min` 和 `max`，`min` 标记最小值，`max` 标记最大值，每次比较相邻两个数，较大者与 `max` 比较，较小者与 `min` 比较，找出最大值和最小值。比较次数为 $1.5N$ 次。

程序示例代码如下：

```
#include <stdio.h>

void GetMaxAndMin(int* arr, int len, int& Max, int& Min)
{
    Max = arr[0];
    Min = arr[0];
    for(int i=2; i< len-1; i=i+2)
    {
        if(NULL==arr[i+1])
        {
            if(arr[i]>Max)
                Max=arr[i];
            if(arr[i]<Min)
                Min=arr[i];
        }
        if(arr[i]>arr[i+1])
        {
            if(arr[i]>Max)
                Max=arr[i];
            if(arr[i+1]<Min)
                Min=arr[i+1];
        }
        if(arr[i]<arr[i+1])
        {
            if(arr[i+1]>Max)
                Max=arr[i+1];
            if(arr[i]<Min)
                Min=arr[i];
        }
    }
}

int main()
{
    int max,min;
    int data[]={8,6,5,2,3,9,4,1,7};
    int num=sizeof(data)/sizeof(data[0]);
    GetMaxAndMin(data,num,max,min);
    printf("Max:%d\n",max);
}
```



```

        printf("Min:%d\n",min);
    return 0;
}

```

程序输出结果:

```

Max:9
Min:1

```

(4) 数组元素移位法。

将数组中相邻的两个数分在一组，每次比较两个相邻的数，将较大值交换至这两个数的左边，较小值放于右边。对大者组扫描一次找出最大值，对小者组扫描一次找出最小值。此种方法需要比较 $1.5N-2N$ 次，但需要改变数组结构。

(5) 分治法。

将数组划分成两半，分别找出两边的最小值、最大值，则最小值、最大值分别是两边最小值的较小者、两边最大值的较大者。此种方法比较次数为 $1.5N$ 次。

程序示例如下:

```

#include <stdio.h>

void GetMaxMin(int a[],int low,int high,int& max,int& min)
{
    int k, max1,min1,max2,min2;
    if(high-low==1||high-low==0)
    {
        a[low]>a[high]? (max = a[low], min = a[high]):(max = a[high], min = a[low]);
    }
    else
    {
        k=(high+low)/2;
        GetMaxMin( a,low,k,max1,min1);
        GetMaxMin( a,k+1,high,max2,min2);
        max=max1>max2? max1:max2;
        min=min1<min2? min1:min2;
    }
}

int main()
{
    int max,min;
    int data[]={8,6,5,2,3,9,4,1,7};
    int num=sizeof(data)/sizeof(data[0]);
    GetMaxMin(data,0,num-1,max,min);
    printf("Max:%d\n",max);
    printf("Min:%d\n",min);
    return 0;
}

```

程序输出结果:

```

Max:9
Min:1

```

13.1.20 如何将数组的后面 m 个数移动为前面 m 个数

有 n 个整数，使前面各数后移 m 个位置，最后 m 个数变成最前面 m 个数。例如，有 10 个数的数组，即 $n=10$ ，它们的值分别是 1,2,3,4,5,6,7,8,9,10，如果取 $m=5$ 的话，经过位置调整

后, 变为 6,7,8,9,10,1,2,3,4,5。

可以通过递归的方法实现调整:

- (1) 将前 m 个元素的顺序颠倒。
- (2) 将后面 $n-m$ 个元素的顺序颠倒。
- (3) 将 n 个元素的顺序全部颠倒。

通过以上 3 个步骤的执行, 就可以把数组的元素颠倒。以上例而言, 第一步以后, 数组顺序变为 5,4,3,2,1,6,7,8,9,10; 第二步以后, 数组顺序变为 5,4,3,2,1,10,9,8,7,6; 第三步以后, 数组顺序变为 6,7,8,9,10,1,2,3,4,5, 正是题目要求的, 此时结束运算。

程序代码示例如下:

```
#include <stdio.h>

void func(int* start, int* end)
{
    while( start < end )
    {
        int temp = *start;
        *start = *end;
        *end = temp;
        ++start;
        --end;
    }
}

void f(int n, int m, int* numbers)
{
    func(numbers, numbers+m-1); // 前 m 个数顺序颠倒
    func(numbers+m, numbers+n-1); // 后 n-m 个数顺序颠倒
    func(numbers, numbers+n-1); // 所有数顺序颠倒
}

int main()
{
    int array[] = {1,2,3,4,5,6,7,8,9,10};
    int len = sizeof(array)/sizeof(array[0]);
    int i;
    f(len,5,array);
    for (i=0;i<len;i++)
        printf("%d ",array[i]);
    return 0;
}
```

程序输出结果:

6 7 8 9 10 1 2 3 4 5

13.1.21 如何计算出序列的前 n 项数据

正整数序列 Q 中的每个元素都至少能被正整数 a 和 b 中的一个整除, 现给定 a 和 b , 如何计算出 Q 中的前几项? 例如, 当 $a=3$, $b=5$, $N=6$ 时, 序列为 3, 5, 6, 9, 10, 12。

可以与归并排序联系起来, 给定两个数组 A 、 B , 数组 A 存放: 3×1 , 3×2 , 3×3 , ... 数组 B 存放: 5×1 , 5×2 , 5×3 , ... 有两个指针 i 、 j , 分别指向 A 、 B 的第一个元素, 取 $\text{Min}(A[i], B[j])$, 并将较小值的指针前移, 然后继续比较。当然, 编程实现的时候, 完全没有必要申请两个数组, 用两个变量就可以了。程序示例如下:

```

#include<stdio.h>

void Generate(int a,int b,int N,int *Q)
{
    int tmpA,tmpB;
    int i=1;
    int j=1;
    for(int k=0;k<N;k++)
    {
        tmpA=a*i;
        tmpB=b*j;
        if(tmpA<=tmpB)
        {
            Q[k]=tmpA;
            i++;
        }
        else
        {
            Q[k]=tmpB;
            j++;
        }
    }
}

int main()
{
    int a[6];
    int i;
    Generate(3,5,6,a);
    for(i=0; i<sizeof(a)/sizeof(a[0]);i++)
        printf("%d ",a[i]);
    printf("\n");
    return 0;
}

```

程序的输出结果:

3 5 6 9 10 12

13.1.22 如何找出数组中只出现一次的数字

一个整型数组里除了一个数字之外，其他的数字都出现了两次。请写程序找出这个只出现一次的数字。要求时间复杂度是 $O(n)$ ，空间复杂度是 $O(1)$ 。

如果本题对时间复杂度没有要求的话，最容易想到的方法就是首先对这个整型数组排序，然后从第一个数字开始遍历，比较相邻的两个数，从而找出这个只出现一次的数字，所以其时间复杂度最快为 $O(n\log n)$ 。

由于时间复杂度与空间复杂度的限制，该方法不可取，所以需要一种更高效的方式。题目强调只有一个数字出现一次，其他的出现了两次，首先想到的是异或运算的性质：任何一个数字异或它自己都等于 0，根据这一特性，如果从头到尾依次异或数组中的每一个数字，因为那些出现两次的数字全部在异或中抵消掉了，所以最终的结果刚好是那个只出现一次的数字。

程序示例如下：

```

#include <stdio.h>

int findNotDouble(int a[], int n)

```

```

{
    int result = a[0];
    int i;
    for(i = 1; i < n; ++i)
        result ^= a[i];
    return result;
}

int main()
{
    int array[] = {1,2,3,2,4,3,5,4,1};
    int len = sizeof(array)/sizeof(array[0]);
    int num = findNotDouble(array,len);
    printf("%d\n",num);
    return 0;
}

```

程序输出结果:

5

引申: 如果题目改为整型数组中除了两个数字之外, 其他的数字都出现了两次, 如何求解这两个只出现一次的数?

在上述解决方案的基础上, 如果能够把原数组分为两个子数组, 在每个子数组中, 包含一个只出现一次的数字, 而其他数字都出现两次, 问题就可以很容易地解决了: 分别对两个子数组按照上述解决方案执行结果。

现在问题的难点就是如何划分为两个符合求解方案的子数组。首先从头到尾依次异或数组中的每一个数字, 因为其他数字都出现了两次, 在异或中全部抵消掉了, 所以最终得到的结果将是两个只出现一次的数字的异或结果。而这两个数字肯定不一样, 那么这个异或结果肯定不为 0, 也就是说在这个结果数字的二进制表示中至少就有一位为 1, 否则就为 0 了。在结果数字中找到第一个为 1 的位的位置, 记为第 N 位, 此时以第 N 位是不是 1 为标准把原数组中的数字分成两个子数组, 第一个子数组中每个数字的第 N 位都为 1, 而第二个子数组的每个数字的第 N 位都为 0。通过这种方法就可以把原数组分成了两个子数组, 每个子数组都包含一个只出现一次的数字, 而其他数字都出现了两次。

程序示例如下:

```

#include <stdio.h>

void findOnce(int data[], int n, int &num1, int &num2)
{
    if (n < 5)
        return;
    int r1 = 0;
    for (int i = 0; i < n; i++)
        r1 ^= data[i];
    int bitNum = 0;
    while ( !(r1 & 0x1) )
    {
        r1 >>= 1;
        ++bitNum;
    }
    int flag = (1 << bitNum);
    num1 = 0;
    num2 = 0;
    for (int j = 0; j < n; j++)

```

```

    {
        if ( data[j] & flag)
            num1 ^= data[j];
        else
            num2 ^= data[j];
    }
}

int main()
{
    int array[] = {1,2,3,2,4,3,5,1};
    int num1, num2;
    findOnce(array, sizeof(array)/sizeof(array[0]), num1, num2);
    printf("%d\n%d\n",num1,num2);
    return 0;
}

```

程序输出结果:

5
4

13.1.23 如何判断一个整数 x 是否可以表示成 n ($n \geq 2$) 个连续正整数的和

假设 x 可以表示成 $n(n \geq 2)$ 个连续正整数的和, 那么数学表达式如下: $x = m + (m+1) + (m+2) + \dots + (m+n-1)$, 其中 m 为分解成的连续整数中最小的那一个, 由于 m 是大于等于 1 的正整数, 可知 $x = (2m+n-1) \times n/2$, 变换之后 $m = (2 \times x/n - n + 1) / 2$, 由 m 的范围可以知道 $(2 \times x/n - n + 1) / 2 \geq 1$, 以上就是 x 和 n 的关系。给定一个 n , 看是否 x 能分解成 n 个连续整数的和, 可以判断是否存在 m , 也就是转换成 $(2 \times x/n - n + 1)$ 是否是偶数的问题。

判断一个数是否是偶数, 是一个比较容易解决的问题, 所以本题的问题就迎刃而解了, 程序示例如下:

```

#include <stdio.h>
#include <math.h>

int main()
{
    int m=0,n=0,start=0,end=0,flag=0;
    float temp=0.0;
    printf("请输入被分解的数:");
    scanf("%d",&m);
    printf("请输入需要被分解的数字的个数:");
    scanf("%d",&n);
    temp=(float)m/n-(float)(n-1)/2;
    if(temp==(int)temp)
    {
        for(flag=1,start=(int) temp,end=start+n;start<end;start++)
            printf("%d ",start);
        printf("\n");
    }
    if(flag==0)
        printf("没有符合条件的数\n");
    return 0;
}

```

程序输出结果:


```
请输入被分解的数:10
请输入需要被分解的数字的个数:4
1 2 3 4
```

13.2 链表

13.2.1 数组和链表的区别是什么

数组与链表是两种不同的数据存储方式。链表的特性是在中间任意位置添加元素、删除元素都非常地快，不需要移动其他的元素。通常对于单链表而言，链表中每一个元素都要保存一个指向下一个元素的指针；而对于双链表，每个元素既要保存一个指向下一个元素的指针，还要保存一个指向上一个元素的指针；循环链表则在最后一个元素中保存一个指向第一个元素的指针。

数组是一组具有相同类型和名称的变量的集合，这些变量称为数组的元素，每个数组元素都有一个编号，这个编号称为数组的下标，可以通过下标来区别并访问这些元素，数组元素的个数也称为数组的长度。

具体而言，数组和链表的区别主要表现在以下几个方面：

(1) 逻辑结构。数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况，即在使用数组之前，就必须对数组的大小进行确定。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费。数组中插入、删除数据项时，需要移动其他数据项。而链表采用动态分配内存的形式实现，可以适应数据动态地增减的情况，需要时可以用 `new/malloc` 分配内存空间，不需要时用 `delete/free` 将已分配的空间释放，不会造成内存空间浪费，且可以方便地插入、删除数据项。

(2) 内存结构。（静态）数组从栈中分配空间，对于程序员方便快速，但是自由度小。链表从堆中分配空间，自由度大，但是申请管理比较麻烦。

(3) 数组中的数据在内存中是顺序存储的，而链表是随机存储的。数组的随机访问效率很高，可以直接定位，但插入、删除操作的效率比较低。链表在插入、删除操作上相对数组有很高的效率，而如果要访问链表中的某个元素，那就得从表头逐个遍历，直到找到所需要的元素为止，所以链表的随机访问效率比数组低。

(4) 链表不存在越界问题，数组有越界问题。数组便于查询，链表便于插入删除，数组节省空间但是长度固定，链表虽然变长但是占了更多的存储空间。

所以，由于数组存储效率高、存储速度快的优点，如果需要频繁访问数据，很少插入删除操作，则使用数组；反之，如果频繁插入删除，则应使用链表。两者各有用处。

13.2.2 何时选择顺序表、何时选择链表作为线性表的存储结构为宜

顺序表按照顺序存储，即数据元素存放在一个连续的存储空间之中，实现顺序存取或（按下标）直接存取。链表按照链接存储，即存储空间一般在程序的运行过程中动态分配与释放，且只要存储器中还有空间，就不会产生存储溢出的问题。

顺序表与链表各有短长，在实际应用中，应根据具体问题的要求和性质来选择使用哪一种存储结构，通常有以下几方面的考虑：

(1) 空间。顺序表的存储空间是静态分配的，链表的存储空间是动态分配的。顺序表的存储密度比链表大，当要求存储的线性表长度变化不大，易于事先确定其大小时，为了节约存储

空间，宜采用顺序表；反之，当线性表长度变化大，难以估计其存储规模时，采用动态链表作为存储结构为好。

(2) 时间。顺序表是随机存取结构，若线性表的操作主要是进行查找，很少做插入和删除操作时，采用顺序表做存储结构为宜；反之，若需要对线性表进行频繁地插入或删除等的操作时，宜采用链表做存储结构。并且，若链表的插入和删除主要发生在表的首尾两端，则采用尾指针表示的单循环链表为宜。

所以，不能笼统地说哪种实现更好，必须根据实际问题的具体需要，并对各个方面的优缺点进行综合评估，才能最终选择一种比较适合的实现方法。

13.2.3 如何使用链表头

在回答这个问题前，首先弄清楚一个概念，什么是结点？简单地说，结点表示的就是数据域与指针域的和，数据域存储数据元素的信息，指针域指示直接后继存储位置，所以结点表示数据元素或数据元素的映象关系。

单链表的开始结点之前附设一个类型相同的结点，称之为头结点，头结点的数据域可以不存储任何信息（也可以存放如线性表的长度等附加信息），头结点的指针域存储指向开始结点的指针（即第一个元素结点的存储位置）。图 13-1 所示为带头结点的单链表。

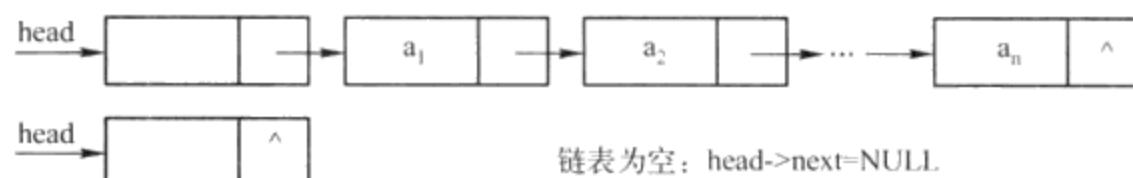


图 13-1 带头结点的单链表

头结点的作用主要有以下两点：

(1) 对带头结点的链表，在表的任何结点之前插入结点或删除表中任何结点，所要做的都是修改前一结点的指针域，因为任何元素结点都有前驱结点。若链表没有头结点，则首元素结点没有前驱结点，在其前插入结点或删除该结点时操作会复杂些。

(2) 对带头结点的链表，表头指针是指向首结点的非空指针，因此空表与非空表的处理是一样的。

在实现运算时，需要动态产生出其头结点，并将其后继指针置为空。

```

void initial_List(node *L)
{
    L=(node*)malloc(sizeof(node));
    L->next=NULL;
}
    
```

需要注意的是，开始结点、头指针、头结点并不是一个概念，它们是有区别的：开始结点是指链表中的第一个结点，也就是没有直接前趋的那个结点，而链表的头指针是指向链表开始结点的指针（没有头结点时），单链表由头指针唯一确定，因此单链表可以用头指针的名字来命名。图 13-2 所示链表的头指针为 head，则称该链表为链表 head，在定义链表变量时可以这样声明：node *head，而头结点是人为地在链表的开始结点之前附加的一个结点。有了头结点之后，头指针指向头结点，无论链表是否为空，头指针总是非空。而且头指针的设置使得对链表的第一个位置上的操作与在表其他位置上的操作一致（都是在某一结点之后）。

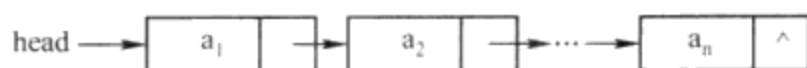


图 13-2 链表 head 结构

13.2.4 如何实现单链表的插入、删除操作

插入运算是将值为 x 的新结点插入到单链表的第 i 个结点的位置上, 即插入到 a_{i-1} 与 a_i 之间。具体算法如下:

- (1) 找到 a_{i-1} 存储位置 p 。
- (2) 生成一个数据域为 x 的新结点 s 。
- (3) 令结点 p 的指针域指向新结点 s 。
- (4) 新结点 s 的指针域指向结点 a_i 。

图 13-3 所示为单链表插入结点示意图。

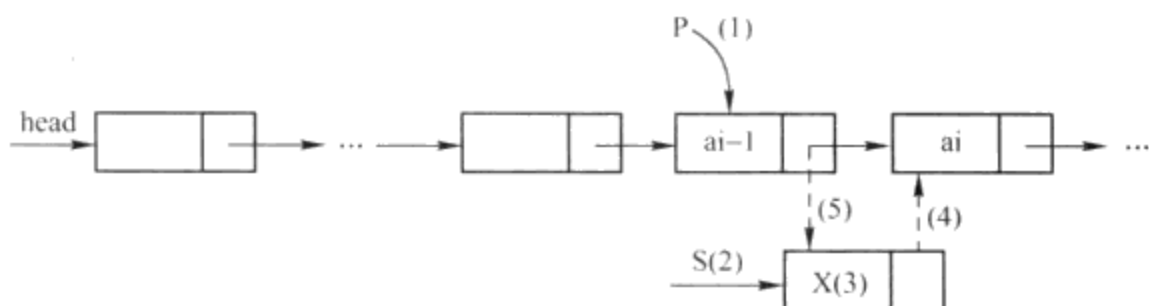


图 13-3 单链表插入结点示意图

单链表插入结点具体算法实现如下:

```
Status InsertList(LinkList head,DataType x,int i)
{
    ListNode *p;
    p=head;
    int j = 1;
    while(p->next && j < i)
    {
        p=p->next;
        ++j;
    }
    if (p==NULL)
    {
        printf("Position Error");
        return ERROR;
    }
    s=(ListNode *)malloc(sizeof(ListNode));
    s->data=x;
    s->next=p->next;
    p->next=s;
    return OK;
}
```

单链表插入算法的时间主要耗费在查找操作 `GetNode`, 即获得结点上, 故单链表插入操作的时间复杂度为 $O(n)$ 。

单链表的删除操作是将单链表的第 i 个结点删去。其具体步骤如下:

(1) 找到 a_{i-1} 的存储位置 p (因为在单链表中结点 a_i 的存储地址是在其直接前趋结点 a_{i-1} 的指针域 `next` 中)。

(2) 令 $p \rightarrow \text{next}$ 指向 a_i 的直接后继结点 (即把 a_i 从链上摘下) a_{i+1} 。

(3) 释放结点 a_i 的空间, 将其归还给“存储池”。

图 13-4 所示为单链表删除结点示意图。

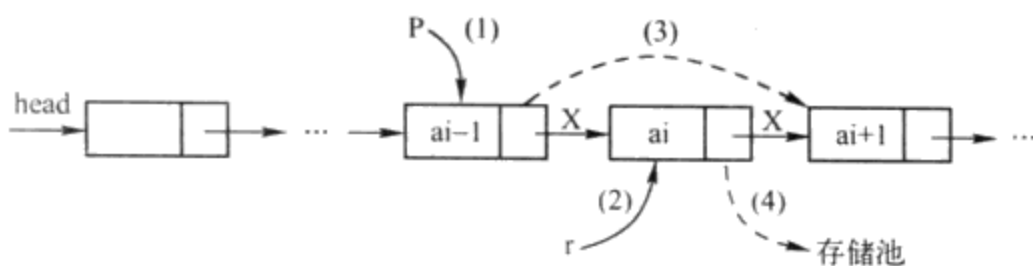


图 13-4 单链表删除结点示意图

单链表删除结点具体算法实现如下：

```

Status DeleteList(LinkList head,int i)
{
    ListNode *p,*r;
    p=head;
    int j = 1;
    while(p->next && j < i)
    {
        p=p->next;
        ++j;
    }
    if (p==NULL || p->next==NULL || j > i)
    {
        printf("Position Error");
        return ERROR;
    }
    r=p->next;
    p->next=r->next;
    free(r);
    return OK;
}

```

设单链表的长度为 n ，则单链表删除第 i 个结点时，必须保证 $1 \leq i \leq n$ ，否则不合法。而当 $i=n+1$ 时，虽然被删结点不存在，但其前趋结点却存在，它是终端结点。因此，被删结点的直接前趋 $*p$ 存在并不意味着被删结点就一定存在，仅当 $*p$ 存在（即 $p \neq \text{NULL}$ ）且 $*p$ 不是终端结点（即 $p \rightarrow \text{next} \neq \text{NULL}$ ）同时满足 $j \leq i$ 时，才能确定被删结点存在。此时，算法的时间复杂度也是 $O(n)$ 。

引申：如何删除单链表的头元素？

要删除头元素，首先需要通过头结点定位头元素，并将头结点指向头元素的下一个元素，然后释放头元素的内存空间。具体代码如下：

```

void RemoveHead(LinkList head)
{
    ListNode *p;
    p=head->next;
    head->next = p->next;
    free(p);
}

```

13.2.5 如何找出单链表中的倒数第 k 个元素

为了找出单链表中的倒数第 k 个元素，最容易想到的方法是首先遍历一遍单链表，求出整个单链表的长度 n ，然后将倒数第 k 个，转换为正数第 $n-k$ 个，接下去遍历一次就可以得到结果。但是该算法需要对链表进行两次遍历，第一次遍历用于求解单链表的长度，第二次遍历用于查找正数第 $n-k$ 个元素。

于是想到了第二种方法,如果沿从头至尾的方向从链表中的某个元素开始,遍历 k 个元素后刚好达到链表尾,那么该元素就是要找的倒数第 k 个元素。根据这一性质,可以设计如下算法:从头结点开始,依次对链表的每一个结点元素进行这样的测试,遍历 k 个元素,查看是否到达链表尾,直到找到那个倒数第 k 个元素为止。此种方法将对同一批元素进行反复多次的遍历,对于链表中的大部分元素而言,都要遍历 k 个元素,如果链表长度为 n ,该算法时间复杂度为 $O(kn)$ 级,效率太低。

存在另外一种更高效的方式,只需要一次遍历即可查找到倒数第 k 个元素。由于单链表只能从头到尾依次访问链表的各个结点,所以如果要找链表的倒数第 k 个元素,也只能从头到尾进行遍历查找。在查找过程中,设置两个指针,让其中一个指针比另一个指针先前移 k 步,然后两个指针同时往前移动。循环直到先行的指针值为 `NULL` 时,另一个指针所指的位置就是所要找的位置。程序代码如下:

```
template<class T>
struct ListNode
{
    T data;
    ListNode* next;
};

template<class T>
ListNode<T>* FindElem(ListNode<T>* head,int k)
{
    ListNode<T>* ptr1,* ptr2;
    ptr1=ptr2=head;
    for(int i=0;i<k;++i) //前移 k 步
    {
        ptr1=ptr1->next;
    }

    while(ptr1!=NULL) //循环检测
    {
        ptr1=ptr1->next;
        ptr2=ptr2->next;
    }
    return ptr2;
}
```

13.2.6 如何实现单链表反转

输入一个链表的头结点,反转该链表,并返回反转后链表的头结点。链表结点定义如下:

```
struct ListNode
{
    int m_nKey;
    ListNode* m_pNext;
}
```

为了正确地反转一个链表,需要调整指针的指向,而与指针操作相关代码总是容易出错的。先举个例子看一下具体的反转过程。例如, l 、 m 、 n 是 3 个相邻的结点,假设经过若干步操作,已经把结点 i 之前的指针调整完毕,这些结点的 `m_pNext` 指针都指向前面一个结点。现在遍历到结点 m ,当然需要调整结点的 `m_pNext` 指针,让它指向结点 l ,但需要注意的是,一旦调整了指针的指向,链表就断开了,因为已经没有指针指向结点 n ,没有办法再遍历到结点 n 了,因此为了避免链表断开,需要在调整 m 的 `m_pNext` 之前要把 n 保存下来。接下来试着

找到反转后链表的头结点，不难分析出反转后链表的头结点是原始链表的尾结点，即 `m_pNext` 为空指针的结点。

具体的实现过程如下：

```
ListNode* ReverseIteratively(ListNode* pHead)
{
    ListNode* pReversedHead=NULL;
    ListNode* pNode=pHead;
    ListNode* pPrev=NULL;
    while(pNode !=NULL)
    {
        ListNode* pNext=pNode->m_pNext;
        if(pNext==NULL)
            pReversedHead=pNode;
        pNode->m_pNext=pPrev;
        pPrev=pNode;
        pNode=pNext;
    }
    return pReversedHead;
}
```

如果本题简化为逆序输出单链表元素，那么递归将是最简单的方法。在递归函数之后输出当前元素，这样能确保输出第 n 个结点的元素语句永远在第 $n+1$ 个递归函数之后执行，也就是说第 n 个元素永远在第 $n+1$ 个元素之后输出，最终先输出最后一个元素，然后是倒数第 2 个、倒数第 3 个，直到输出第一个元素位置。具体实现过程如下：

```
void PrintReverseLink(ListNode* head)
{
    if (head->Next != null)
    {
        PrintReverseLink (head->m_pNext);
        printf("%d\n",head->m_pNext->m_nKey);
    }
}
```

本题不是要求逆序输出，而是需要把单链表逆序，所以在用递归思想的时候，还需要处理递归后的逻辑问题。具体而言，是在反转当前结点之前先调用递归函数反转后续结点，不过该方法存在一个问题，就是反转后的最后一个结点会形成一个环，所以必须将函数返回的结点的 `m_pNext` 域设置为 `NULL`，同时考虑到链表反转时需要改变 `head` 指针，所以在进行参数传递时，需要传递引用。

具体的实现过程如下：

```
ListNode* Reverse(ListNode* p, ListNode* & head)
{
    if(p == NULL || p->m_pNext == NULL)
    {
        head = p;
        return p
    }
    else
    {
        ListNode* temp = Reverse (p->m_pNext, head);
        tmp->m_pNext = p;
        return p;
    }
}
```

需要注意的是，当单链表有环时，就会无法反转，因为如果单链表有环，则存在两个结点

指向同一结点的情况。如果反转就变成一个结点指向两个了，而这对于单链表是不可能的。

13.2.7 如何从尾到头输出单链表

```
struct ListNode
{
    int m_nKey;
    ListNode *m_pNext;
};
```

从头到尾输出单链表比较简单，于是很自然地想把链表中链接结点的指针反转过来，改变链表的方向，然后就可以从尾到头输出了，但该方法需要额外的操作，是否还有更好的方法呢？答案是肯定的。

接下来的想法是从头到尾遍历链表，每经过一个结点的时候，把该结点放到一个栈中。当遍历完整个链表后，再从栈顶开始输出结点的值，此时输出的结点的顺序已经反转过来了。该方法虽然没有只需要遍历一遍链表，但是需要维护一个额外的栈空间，实现起来会比较麻烦。

是否还能有更高效的方法？于是我们想到了第三种方法，既然想到了栈来实现这个函数，而递归本质上就是一个栈结构，于是很自然地又想到了用递归来实现。要实现反过来输出链表，每访问到一个结点的时候，先递归输出它后面的结点，再输出该结点自身，这样链表的输出结果就反过来了。

具体实现如下：

```
void PrintListReversely(ListNode* pListHead)
{
    if(pListHead != NULL)
    {
        PrintListReversely(pListHead->m_pNext);
    }
    printf("%d", pListHead->m_nKey);
}
```

该题还有两个常见的变体：

- (1) 从尾到头输出一个字符串。
 - (2) 定义一个函数求字符串的长度，要求该函数体内不能声明任何变量。
- 对于这两个变体的解答，都可以参考本题的实现方式，在此就不再赘述了。

13.2.8 如何寻找单链表的中间结点

最容易想到的思路是首先求解单链表的长度 `length`，然后遍历 `length/2` 的距离即可查找到单链表的中间结点，但是此种方法需要遍历两次链表，即第一次遍历求解单链表的长度，第二次遍历根据索引获取中间结点。

如果是双向链表，可以首尾并行，利用两个指针一个从头到尾，一个从尾到头，当两个指针相遇的时候就找到中间元素。以此思想为基础，如果是单链表也可以采用双指针的方式来实现中间结点的快速查找。

第一步，有两个指针同时从头开始遍历。第二步，一个快指针一次走两步，一个慢指针一次走一步。第三步，快指针先到链表尾部，而慢指针则恰好到达链表中部（快指针到链表尾部，当链表长度为奇数时，慢指针指向的即是链表中间指针；当链表长度为偶数时，慢指针指向的结点和慢指针指向结点的下一个结点都是链表的中间结点）。

具体实现如下：

```
void SearchMid(node* head, node* mid)
```

```

{
    node* temp = head;
    while(head->next->next != NULL)
    {
        head = head->next->next;
        temp = temp->next;
        mid=temp;
    }
}

```

13.2.9 如何进行单链表排序

最容易想到的排序算法是冒泡排序法，所以首先用冒泡排序法进行单链表的排序。程序示例如下：

```

#include<stdio.h>
#include<stdlib.h>

typedef struct node
{
    int data;
    struct node *next;
}linklist;

linklist *head=NULL;

linklist* CreateList(int* arr, int len)
{
    int data;
    linklist *pCurrent,*rear;
    head = (linklist*)malloc(sizeof(linklist));
    rear = head;

    int count = 0;
    while (count<len)
    {
        pCurrent = (linklist*)malloc(sizeof(linklist));
        pCurrent->data = arr[count];
        rear->next = pCurrent;
        rear= pCurrent;
        count++;
    }
    rear->next = NULL;
    return head;
}

void ShowList(linklist *p)
{
    while(p)
    {
        printf("%d ",p->data);
        p = p->next;
    }
    printf("\n");
}

void BubbleSortList(linklist *p) //链表冒泡排序

```

```

{
    linklist *_temp = p->next;
    linklist *_node = p->next;
    int temp;
    for (; _temp->next; _temp = _temp->next)
    {
        for (_node = p->next; _node->next; _node = _node->next)
        {
            if (_node->data > _node->next->data)
            {
                temp = _node->data;
                _node->data = _node->next->data;
                _node->next->data = temp;
            }
        }
    }
}

int main()
{
    int array[] = {3,4,5,1,2,-1,7};
    CreateList(array,sizeof(array)/sizeof(array[0]));
    BubbleSortList(head);
    ShowList(head->next);
    return 0;
}

```

程序输出结果:

-1 1 2 3 4 5 7

在各种排序算法中,冒泡排序并非最高效的,对链表这一特定数据结构而言,最好使用归并排序算法。而堆排序、快速排序这些在数组排序时性能非常好的算法,用在只能“顺序访问”的链表中却不尽如人意,但是归并排序却可以,它不仅保持了 $O(n\log n)$ 的时间复杂度,而且它的空间复杂度也从 $O(n)$ 降到了 $O(1)$,除此之外,归并排序是分治法的实现。具体实现过程如下:

```

#include <iostream>
#define MAXSIZE 1024
#define LENGTH 8

using namespace std;

typedef struct
{
    int r[MAXSIZE+1];
    int length;
} SqList;

void Merge(SqList SR, SqList &TR, int i, int m, int n)
{
    int j, k;
    for(j=m+1, k=i; i<=m && j<=n; ++k)
    {
        if(SR.r[i]<=SR.r[j])
            TR.r[k]=SR.r[i++];
        else
            TR.r[k]=SR.r[j++];
    }
}

```

```

        while(i<=m)
            TR.r[k++]=SR.r[i++];
        while(j<=n)
            TR.r[k++]=SR.r[j++];
    }

void MSort(SqList SR,SqList &TR1,int s, int t)
{
    int m;
    SqList TR2;
    if(s==t)
        TR1.r[s]=SR.r[t];
    else
    {
        m=(s+t)/2;
        MSort(SR,TR2,s,m);
        MSort(SR,TR2,m+1,t);
        Merge(TR2,TR1,s,m,t);
    }
}

void MergeSort(SqList &L)
{
    MSort(L,L,1,L.length);
}

int main()
{
    int i;
    SqList L={{0,49,38,65,97,76,13,27},LENGTH};
    MergeSort(L);
    for(i=1;i<=L.length;i++)
        cout<<L.r[i]<<" ";
    cout<<endl;
    return 0;
}

```

程序输出结果:

0 13 27 38 49 65 76 97

13.2.10 如何实现单链表交换任意两个元素（不包括表头）

对于单链表而言，假设交换的结点为 A 与 B，那么需要交换 A 与 B 的 next 指针以及 AB 直接前驱的 next 指针。需要注意的特殊情况：当 A 与 B 相邻时，此时需要做特殊处理；如果 A 与 B 元素相同，就没有必要交换；如果 A 与 B 结点中有一个是表头，也不交换。

程序示例如下：

```

struct node
{
    int data;
    node* next;
};

//返回前驱结点
node* FindPre(node*head, node*p)
{
    node *q = head;

```



```

while(q)
{
    if(q->next == p)
        return q;
    else
        q = q->next;
}
return NULL;
}

node* Swap(node *head, node *p, node *q)
{
    if ( head == NULL || p == NULL || q == NULL )
    {
        cout<<"invalid parameter: NULL"<<endl;
        return head;
    }
    if (p->data==q->data)
        return head;
    if (p->next == q)
    {
        node* pre_p = FindPre(head, p);
        pre_p->next = q;
        p->next = q->next;
        q->next = p;
    }
    else if (q->next == p)
    {
        node* pre_q = FindPre(head, q);
        pre_q->next = p;
        q->next = p->next;
        p->next = q;
    }
    else if ( p != q)
    {
        node* pre_p = FindPre(head, p);
        node* pre_q = FindPre(head, q);
        node* after_p = p->next;
        p->next = q->next;
        q->next = after_p;
        pre_p->next = q;
        pre_q->next = p;
    }
    return head;
}

```

13.2.11 如何检测一个较大的单链表是否有环

单链表有环是指单链表中某个结点的 next 指针域指向的是链表中在它之前的某一个结点, 这样在链表的尾部形成一个环形结构。检测单链表是否有环, 一般有以下几种方法。

方法一: 定义一个指针数组, 初始化为空指针, 从链表的头指针开始往后遍历, 每次遇到一个指针就跟指针数组中的指针相比较, 若没有找到相同的指针, 说明这个结点是第一次访问, 还没有形成环, 将这个指针添加到指针数组中去。若在指针数组中找到了同样的指针, 说明这个结点已经被访问过了, 于是就形成了环。

方法二：定义两个指针 fast 与 slow，两者的初始值都指向头，slow 每次前进一步，fast 每次前进两步，两个指针同时向前移动，快指针每移动一次都要跟慢指针比较，直到当快指针等于慢指针为止，就证明这个链表是带环的单向链表，否则证明这个链表是不带环的循环链表（fast 先行到达尾部为 NULL，则为无环链表）。

```
struct listtype
{
    int data;
    struct listtype *next;
};
typedef struct listtype *list;
int IsLoop(list sll)
{
    list fast=sll;
    list slow=sll;
    if(fast==NULL)
    {
        return -1;
    }
    while(fast&&fast->next)
    {
        fast=fast->next->next;
        slow=slow->next;
        if(fast==slow)
        {
            return 1;
        }
    }
    return !(fast==NULL)||fast->next==NULL;
}
```

方法三：通过使用 STL 库中的 map 表进行映射。首先定义 map<node*,int>m; 将一个 node* 指针映射成数组的下标，并赋值为一个 int 类型的数值。然后从链表的头指针开始往后遍历，每次遇到一个指针 p，就判断 m[p] 是否为 0。如果为 0，则将 m[p] 赋值为 1，表示该结点是第一次访问；而如果 m[p] 的值为 1，则说明这个结点已经被访问过一次了，于是就形成了环。

程序代码示例如下：

```
map<node*,int>m;
bool IsLoop(node *head)
{
    if(!head)
        return false;
    node *p=head;
    while(p)
    {
        if(m[p]==0) // 默认值都是 0
            m[p]=1;
        else if(m[p]==1)
            return true;
        p=p->next;
    }
}
```

如果单链表有环，按照方法二的思路，走得快的指针 fast 若与走得慢的指针 slow 相遇时，slow 指针肯定没有遍历完链表，而 fast 指针已经在环内循环了 n 圈 ($1 \leq n$)。假设 slow 指针走了 s 步，则 fast 指针走了 2s 步（fast 步数还等于 s 加上在环上多转的 n 圈），设环长为

r, 则满足如下关系表达式:

$$2s = s + nr$$

$$s = nr$$

设整个链表长为 L , 入口环与相遇点距离为 x , 起点到环入口点的距离为 a 。则满足如下关系表达式:

$$a + x = nr$$

$$a + x = (n-1)r + r = (n-1)r + L - a$$

$$a = (n-1)r + (L - a - x)$$

$(L - a - x)$ 为相遇点到环入口点的距离, 从链表头到环入口点等于 $(n-1)$ 循环内环+相遇点到环入口点, 于是从链表头与相遇点分别设一个指针, 每次各走一步, 两个指针必定相遇, 且相遇第一点为环入口点。

程序代码如下:

```
list* FindLoopPort(list *head)
{
    list *slow = head, *fast = head;
    while ( fast && fast->next )
    {
        slow = slow->next;
        fast = fast->next->next;
        if ( slow == fast ) break;
    }

    if (fast == NULL || fast->next == NULL)
        return NULL;

    slow = head;
    while (slow != fast)
    {
        slow = slow->next;
        fast = fast->next;
    }

    return slow;
}
```

13.2.12 如何判断两个单链表（无环）是否交叉

单链表相交是指两个链表存在完全重合的部分（注意，不是交叉到一个点）。判断两个单链表是否交叉，一般有两种方法：1）将这两个链表首尾相连，然后检测看这个链表是否存在环，如果存在，则两个链表相交，而检测出来的依赖环入口即为相交的第一个点。2）利用链表相交的性质，如果两个链表相交，那么两个链表从相交点到链表结束都是相同的结点，必然是 Y 字形，所以判断两个链表的最后一个结点是不是相同即可。即先遍历一个链表，直到尾部，再遍历另外一个链表，如果也可以走到同样的结尾点，则两个链表相交，这时记下两个链表的长度 $length$ ，再遍历一次，长链表结点先出发前进 $(lengthMax-lengthMin)$ 步，之后两个链表同时前进，每次一步，相遇的第一点即为两个链表相交的第一个点。

程序代码实现如下:

```
bool IsIntersect(Node* list1, Node* list2, Node*& value)
{
    value = NULL;
```

```

if( list1 == NULL || list2 == NULL )
    return false;
Node *temp1 = list1, *temp2 = list2;
int size1 = 0, size2 = 0;
while( temp1->next )
{
    temp1 = temp1->next;
    ++size1;
}
while( temp2->next )
{
    temp2 = temp2->next;
    ++size2;
}

if( temp1 == temp2 )
{
    if( size1 > size2 )
        while( size1 - size2 > 0 )
        {
            list1 = list1->next;
            --size1;
        }
    if( size2 > size1 )
        while( size2 - size1 > 0 )
        {
            list2 = list2->next;
            --size2;
        }
    while( list1 != list2 )
    {
        list1 = list1->next;
        list2 = list2->next;
    }
    value = list1;
    return true;
}
else
    return false;
}

```

13.2.13 如何删除单链表中的重复结点

一个没有排序的链表，如 $list = \{a, l, x, b, e, f, f, e, a, g, h, b, m\}$ ，请去掉重复项，并保留原顺序，以上链表去掉重复项后为 $newlist = \{a, l, x, b, e, f, g, h, m\}$ ，请写出一个高效算法。

方法一：递归求解。

```

link delSame(link head)
{
    link pointer, temp = head;
    if (head->next == NULL)
        return head;
    head->next = delSame(head->next);
    pointer = head->next;
    while (pointer != NULL)
    {

```

```

        if (head->number == pointer->number)
        {
            temp->next = pointer->next;
            free(pointer);
            pointer = temp->next;
        }
        else
        {
            pointer = pointer->next;
            temp = temp->next;
        }
    }
    return head;
}

```

采用递归方法效率不够高效，于是想到了方法二的 hash 法。

方法二：使用 hash 法，具体过程如下。

(1) 建立一个 hash_map，key 为链表中已经遍历的结点内容，开始时为空。

(2) 从头开始遍历链表中的结点。

1) 如果结点内容已经在 hash_map 中存在，则删除此结点，继续向后遍历。

2) 如果结点内容不在 hash_map 中，则保留此结点，将结点内容添加到 hash_map 中，继续向后遍历。

13.2.14 如何合并两个有序链表（非交叉）

合并两个有序链表，一般可以采用递归和非递归两种方式实现。

首先看递归方式，设两个链表的头结点分别为 head1、head2，如果 head1 为空，则直接返回 head2；如果 head2 为空，则直接 head1。如果 head1 链表的第一个数据小于 head2 链表的第一个数据，则把 head1 链表的第一个元素存储到新合并的链表中，递归遍历去除第一个元素的 head1 链表和整个 head2 链表。如果 head1 链表的第一个元素大于或等于 head2 链表的第一个元素，则把 head2 链表的第一个元素存储到新合并的链表中，递归遍历整个 head1 链表，去除第一个元素的 head2 链表。具体过程如下：

```

Node * MergeRecursive(Node *head1 , Node *head2)
{
    if ( head1 == NULL )
        return head2 ;
    if ( head2 == NULL )
        return head1 ;
    Node *head = NULL ;
    if ( head1->data < head2->data )
    {
        head = head1 ;
        head->next = MergeRecursive(head1->next,head2);
    }
    else
    {
        head = head2 ;
        head->next = MergeRecursive(head1,head2->next);
    }
    return head ;
}

```

使用非递归的方式时，分别用指针 head1、head2 来遍历两个链表，如果当前 head1 指向

的数据小于 head2 指向的数据, 则将 head1 指向的结点归入合并后的链表中; 否则将 head2 指向的结点归入合并后的链表中。如果有...一个链表遍历结束, 则把未结束的链表连接到合并后的链表尾部。具体过程如下:

```
Node* Merge(Node *head, Node *head1, Node *head2)
{
    Node *tmp=head;
    while(NULL != head1 && NULL != head2)
    {
        if(head1->data<head2->data)
        {
            tmp->next=head1;
            tmp=head1;
            head1=head1->next;
        }
        else
        {
            tmp->next=head2;
            tmp=head2;
            head2=head2->next;
        }
    }
    if(NULL != head1)
    {
        tmp->next=head1;
    }
    if(NULL != head2)
    {
        tmp->next=head2;
    }
    return tmp;
}
```

13.2.15 什么是循环链表

循环链表 (Circular Linked List) 是一种首尾相接的链表, 它与单链表的唯一区别在于对尾结点的处理, 因为在单链表中尾结点的指针域 NULL 改为指向头结点就得到了单循环链表。

在单循环链表上的操作基本上与非循环链表相同, 只是将原来判断指针是否为 NULL 变为是否是头指针而已, 没有其他较大的变化。图 13-5 所示为带头结点的单循环链表。

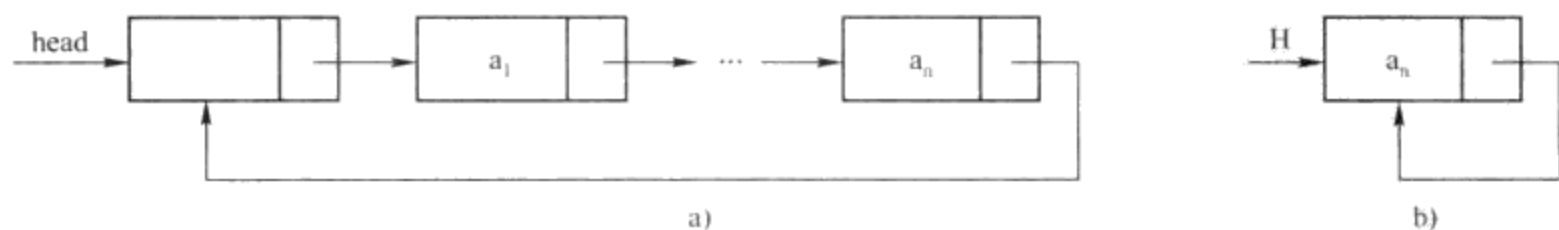


图 13-5 带头结点的单循环链表

a) 非空表 b) 空表

对于单链表只能从头结点开始遍历整个链表, 而对于单循环链表则可以从表中任意结点开始遍历整个链表。因为有时需要对链表常做的操作是在表尾、表头进行, 此时可以改变一下链表的标识方法, 不用头指针而用一个指向尾结点的指针 rear 来标识, 可以使得操作效率得以提高。例如, 用尾指针 rear 表示的单循环链表查找开始结点 a1 和尾结点 an 就很方便, 此时查找时间复杂度都为 $O(1)$ 。

例如, 对两个单循环链表 H1、H2 的连接操作, 是将 H2 的第一个数据结点接到 H1 的尾

结点, 若用头指针标识, 则需要找到第一个链表的尾结点, 其时间复杂性为 $O(n)$; 而链表若用尾指针 $R1$ 、 $R2$ 来标识, 则时间性能为 $O(1)$ 。操作如下:

```
p=R1->next;           //保存 R1 的头结点指针
R1->next=R2->next->next; //头尾连接
free(R2->next);        //释放第二个表的头结点
R2->next=p;            //组成循环链表
```

具体过程如图 13-6 所示。

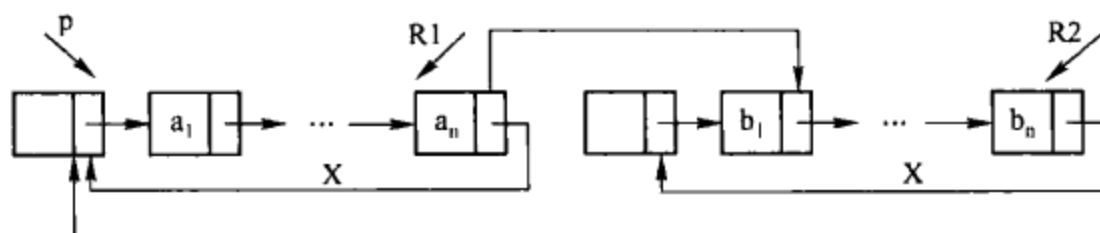


图 13-6 单循环链表连接

仅设尾指针 $rear$ 的单循环链表的实现如下:

```
#include<stdio.h>
#include<stdlib.h>

typedef struct node
{
    int data;
    struct node *next;
}linklist;
linklist *rear=NULL;

linklist* CreateSingleLoopList() //单循环链表的实现
{
    int _data;
    linklist *pCurrent,*head;
    head = (linklist*)malloc(sizeof(linklist));
    scanf("%d",&_data);
    rear->data = _data;
    rear->next = head;
    head->next = rear;
    scanf("%d",&_data);
    while (_data != -1)
    {
        pCurrent = (linklist*)malloc(sizeof(linklist));
        pCurrent->data = _data;
        rear->next = pCurrent;
        pCurrent->next = head;
        rear = pCurrent;
        scanf("%d",&_data);
    }
    return rear;
}

void GetRearHead(linklist *p) //取开始结点和尾结点
{
    printf("%d",p->data); //终端结点 an
    printf("%d",p->next->next->data); //开始结点 a1
}

int main()
```

```

{
    rear = (linklist*)malloc(sizeof(linklist));
    CreateSingleLoopList();
    GetRearHead(rear);
    return 0;
}

```

13.2.16 如何实现双向链表的插入、删除操作

循环单链表的出现, 虽然能够实现从任一结点出发沿着链能找到其前驱结点, 但是时间复杂度为 $O(n)$ 。如果希望从链表中快速确定某一个结点的前驱, 另一个解决方法就是在单链表的每个结点里再增加一个指向其前驱的指针域 pr 。这样形成的链表中就有两条方向不同的链, 被称为双向链表 (Double Linked List)。双向链表简称双链表, 它是由头指针 $head$ 唯一确定的。带头结点的双链表的某些运算变得方便。将头结点和尾结点链接起来, 为双循环链表。

带头结点的双链表的结点结构如图 13-7 所示。



图 13-7 带头结点的双向链表

双链表的形式描述:

```

typedef struct dlistnode
{
    DataType data;
    struct dlistnode *prior,*next;
}DListNode;
typedef DListNode *DLinkedList;
DLinkedList head;

```

由于双链表的对称性, 在双链表中能方便地完成各种插入、删除操作。

在双向链表第 i 个结点 p 之前插入一个新的结点, 则指针的变化情况如图 13-8 所示。

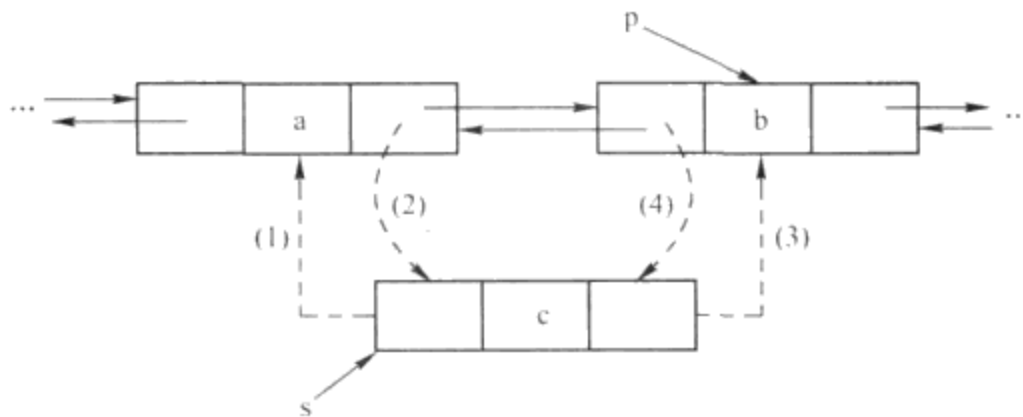


图 13-8 双向链表插入操作

具体实现代码如下:

```

void DInsertBefore()
{
    //在带头结点的双链表中, 将值为 x 的新结点插入 *p 之前, 设 p 不等于 NULL
    DListNode *s=malloc(sizeof(DListNode));
    s->data=x;
    s->prior=p->prior;
    s->next=p;
    p->prior->next=s;
    p->prior=s;
}

```

双链表上删除结点*p 自身的操作如图 13-9 所示。

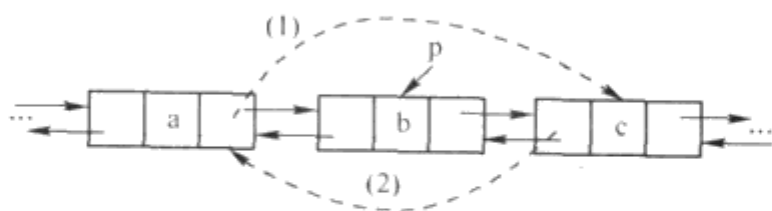


图 13-9 双向链表的删除操作

具体实现代码如下：

```
void DDeleteNode(DListNode *p)
{
    //在带头结点的双链表中，删除结点*p，设*p 为非终端结点
    p->prior->next=p->next;
    p->next->prior=p->prior;
    free(p);
}
```

需要注意的是，与单链表上的插入和删除操作不同的是，在双链表中插入和删除必须同时修改两个方向上的指针。上述两个算法的时间复杂度均为 $O(1)$ 。

13.2.17 为什么在单循环链表中设置尾指针比设置头指针更好

尾指针是指向终端结点的指针，用它来表示单循环链表可以使得查找链表的开始结点和终端结点都很方便，设一带头结点的单循环链表，其尾指针为 *rear*，则开始结点和终端结点的位置分别是 *rear*→*next*→*next* 和 *rear*，查找时间都是 $O(1)$ 。

若用头指针来表示该链表，则查找终端结点的时间为 $O(n)$ 。

13.2.18 如何删除结点的前驱结点

假设在长度大于 1 的单循环链表中，既无头结点也无头指针，*s* 为指向链表中某个结点的指针，如何删除结点*s 的直接前驱结点？

已知指向这个结点的指针是*s，那么要删除这个结点的直接前趋结点，首先需要找到一个结点，它的指针域是指向*s 的，然后再把这个结点删除。具体过程如下：

```
void DeleteNode( ListNode *s)
{
    ListNode *p, *q;
    p=s;
    while( p->next->next!=s)
    {
        q=p;
        p=p->next;
    }
    q->next=s;
    free(p);
}
```

13.2.19 如何实现双向循环链表的删除与插入操作

双向循环链表是双向链表和循环链表的综合。循环链表与单链表相同，是一种链式的存储结构。所不同的是，循环链表的最后一个结点的指针是指向该循环链表的第一个结点或表头结点，从而构成一个环形的链。在双向链表中，结点除含有数据域外，更有两个链域，一个存储

直接后继结点地址，一般称为右链域；一个存储直接前驱结点地址，一般称为左链域。

与单链表类似，双向链表通常也是用头指针标识，也可以带头结点和做成循环结构，图 13-10 所示为带头结点的双向循环链表示意图。

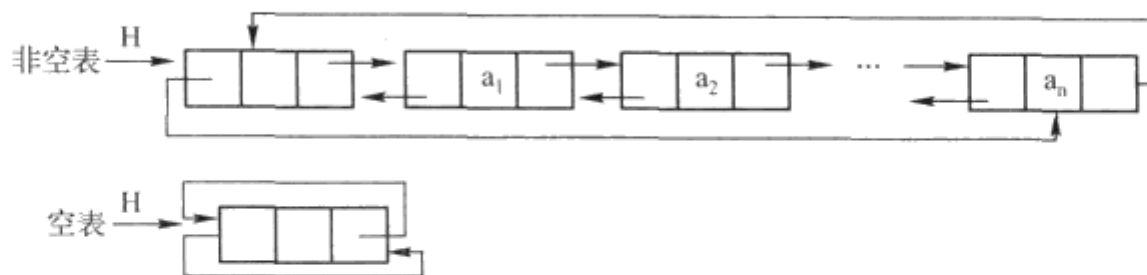


图 13-10 带头结点的双向循环链表

通过某结点的指针 p 即可以直接得到它的后继结点的指针 $p \rightarrow next$ ，也可以直接得到它的前驱结点的指针 $p \rightarrow prior$ ，所以在有些操作中需要找前驱时，则必须再使用循环。例如，结点的删除操作。

设 p 是指向双向循环链表中的某一结点，即 p 是该结点的指针，则 $p \rightarrow prior \rightarrow next$ 表示的是 $*p$ 结点之前驱结点的后继结点的指针，即与 p 相等，而 $p \rightarrow next \rightarrow prior$ 表示的是 $*p$ 结点之后继结点的前驱结点的指针，也与 p 相等。

双向链表中结点的插入：设 p 指向双向链表中某结点， s 指向待插入的值为 x 的新结点，将 $*s$ 插入到 $*p$ 的前面，插入过程如图 13-11 所示。

操作如下：

- (1) $s \rightarrow prior = p \rightarrow prior$ 。
- (2) $p \rightarrow prior \rightarrow next = s$ 。
- (3) $s \rightarrow next = p$ 。
- (4) $p \rightarrow prior = s$ 。

指针操作的顺序不是唯一的，但也不是任意的，第一步操作必须要放到第四步操作之前完成，否则 $*p$ 的前驱结点的指针就丢掉了。

双向链表中结点的删除：设 p 指向双向链表中某结点，删除 $*p$ 。操作过程如图 13-12 所示。

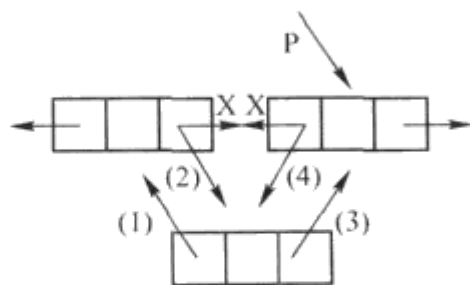


图 13-11 双向链表插入操作

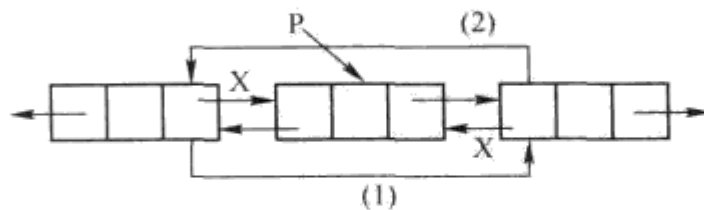


图 13-12 双向链表删除操作

操作如下：

- (1) $p \rightarrow prior \rightarrow next = p \rightarrow next$ 。
- (2) $p \rightarrow next \rightarrow prior = p \rightarrow prior$ 。
- (3) $free(p)$ 。

13.2.20 如何在不知道头指针的情况下将结点删除

在单链表、双链表和单循环链表中，若仅知道指针 p 指向某结点，不知道头指针，能否将

结点 *p 从相应的链表中删去？若可以，其时间复杂度各为多少？

分别讨论 3 种链表的情况。

(1) 单链表。当知道指针 p 指向某结点时，能够根据该指针找到其直接后继，但是由于不知道其头指针，所以无法访问到 p 指针指向的结点的直接前趋。因此无法删去该结点。

(2) 双链表。由于这样的链表提供双向链接，因此根据已知结点可以查找到其直接前趋和直接后继，从而可以删除该结点。其时间复杂度为 $O(1)$ 。

(3) 单循环链表。根据已知结点位置，可以直接得到其后相邻的结点位置（直接后继），又因为是循环链表，所以可以通过查找得到 p 结点的直接前趋。因此，可以删去 p 所指的结点。其时间复杂度应为 $O(n)$ 。

13.3 字符串

一般而言，代码的实现相对容易，提高效率就比较困难，效率往往成为软件开发的瓶颈，而与字符串有关的数据结构与算法历来都是程序员面试笔试的重点。

13.3.1 如何统计一行字符串中有多少个单词

单词的数目可以由空格出现的次数决定（连续的若干个空格作为出现一次空格；一行开头的空格不统计在内）。如果测出某一个字符为非空格，而它的前面的字符是空格，则表示“新的单词开始了”，此时使单词数 count 累加 1。如果当前字符为非空格而其前面的字符也是非空格，则意味着仍然是原来那个单词的继续，count 不应再累加 1。前面一个字符是否空格可以从 word 的值看出来，若 word 等于 0，则表示前一个字符是空格；如果 word 等于 1，意味着前一个字符为非空格。

程序示例如下：

```
#include <stdio.h>
#define BUFFERSIZE 1024

int main( )
{
    char string[BUFFERSIZE];
    int i,count=0,word=0;
    char c;
    gets(string);
    for(i=0;(c=string[i])!='\0';i++)
    {
        if(c==' ')
            word=0;
        else if(word==0)
        {
            word=1;
            count++;
        }
    }
    printf("一共有单词 %d 个\n",count);
    return 0;
}
```

程序输出结果：

i am hehao
一共有单词 3 个

上例中(c=string[i])!='\0'的作用是先将字符数组的某一元素(一个字符)赋给字符变量 c, 此时赋值表达式的值就是该字符, 然后再判定它是否是结束符。

13.3.2 如何将字符串逆序

给定一个字符串 s, 将 s 中的字符顺序颠倒过来, 如 s="abcd", 逆序后变成 s="dcba"。可以采用多种方法对字符串进行逆序, 以下将对其中的一些方法进行分析。

(1) 普通逆序。

直接分配一个与原字符串等长的字符数组, 然后反向拷贝即可。程序示例如下:

```
#include<stdio.h>

char *Reverse(char *s)
{
    char *q = s;
    while(*q++)
        ;
    q -= 2;
    char *p = new char[sizeof(char) * (q - s + 2)];
    char *r = p;
    // 逆序存储
    while(q >= s)
        *p++ = *q--;
    *p = '\0';
    return r;
}

int main()
{
    char a[]="abcd";
    int len = sizeof(a)/sizeof(a[0]);
    printf("%s\n",Reverse(a));
    return 0;
}
```

程序输出结果

dcba

(2) 原地逆序。

原地逆序不允额外分配空间, 就是将字符串两边的字符逐个交换。例如, 给定字符串 "abcd", 逆序的过程分别是交换字符 a 和 d, 交换字符 b 和 c。实现原地逆序的方式有以下 3 种。

1) 设置两个指针, 分别指向字符串的头部和尾部, 然后交换两个指针所指的字符, 并向中间移动指针直到交叉。

程序示例如下:

```
#include<stdio.h>

char *Reverse(char *s)
{
    char *p = s;
    char *q = s;
```

```

    while(*q)
        ++q;
    q--;
    while(q > p)
    {
        char t = *p;
        *p++ = *q;
        *q-- = t;
    }
    return s;
}

int main( )
{
    char a[]="abcd";
    int len = sizeof(a)/sizeof(a[0]);
    printf("%s\n",Reverse(a));
    return 0;
}

```

程序输出结果

dcba

2) 递归, 需要给定逆序的区间, 调用方法 `Reverse(s, 0, strlen(s))`, 对字符串 `s` 在区间 `left` 和 `right` 之间进行逆序, 程序代码示例如下。

```

char *Reverse( char *s, int left, int right )
{
    if(left >= right)
        return s;
    char t = s[left];
    s[left] = s[right];
    s[right] = t;
    Reverse(s, left + 1, right - 1);
}

```

3) 非递归, 同样指定逆序区间, 和方法 (1) 没有本质区别, 一个使用指针, 一个使用下标。

```

char *Reverse( char *s, int left, int right )
{
    while( left < right )
    {
        char t = s[left];
        s[left++] = s[right];
        s[right--] = t;
    }
    return s;
}

```

(3) 不允许临时变量的原地逆序。

原地逆序虽然没有额外分配空间, 但还是使用了临时变量, 占用了额外的空间。如果不允许使用额外空间, 主要有以下两种方法: 第一种是异或操作, 因为异或操作可以交换两个变量而无需借助第三个变量; 第二种是使用字符串的结束符 `'\0'` 所在的位置作为交换空间, 但这有个局限, 只适合以 `'\0'` 结尾的字符串, 对于不支持这种字符串格式的语言, 就不能使用了。

1) 异或操作。

```
#include<stdio.h>
```

```

char* Reverse(char* s)
{
    char* r = s;
    char* p = s;
    while (*(p + 1) != '\0')
        ++p;
    while (p > s)
    {
        *p = *p ^ *s;
        *s = *p ^ *s;
        *p = *p-- ^ *s++;
    }
    return r;
}

```

```

int main( )
{
    char a[]="abcd";
    int len = sizeof(a)/sizeof(a[0]);
    printf("%s\n",Reverse(a));
    return 0;
}

```

程序输出结果:

dcba

2) 使用字符串结束符'\0'所在的位置作为交换空间。

程序示例如下:

```
#include<stdio.h>
```

```

char* Reverse(char* s)
{
    char* r = s;
    char* p = s;
    while (*p != '\0')
        ++p;
    char* q = p - 1;
    while (q > s)
    {
        *p = *q;
        *q-- = *s;
        *s++ = *p;
    }
    *p = '\0';
    return r;
}

```

```

int main( )
{
    char a[]="abcd";
    int len = sizeof(a)/sizeof(a[0]);
    printf("%s\n",Reverse(a));
    return 0;
}

```

程序输出结果:

dcba

(4) 按单词逆序。

给定一个字符串,按单词将该字符串逆序。例如。给定“This is a sentence”,则输出是“sentence a is This”,为了简化问题,字符串中不包含标点符号。一共分两个步骤,第一步先按单词逆序得“sihT si a ecnetnes”,第二步整个句子逆序得到“sentence a is This”。

对于步骤一,关键是如何确定单词,这里以空格为单词的分界。当找到一个单词后,就可以使用上面讲过的方法将这个单词进行逆序,当所有的单词都逆序以后,将整个句子看做一个整体(即一个大的包含空格的单词)再逆序一次即可。

程序示例如下:

```
#include<stdio.h>
void ReverseWord(char* p, char* q)
{
    while(p < q)
    {
        char t = *p;
        *p++ = *q;
        *q-- = t;
    }
}
char* Reverse(char *s)
{
    char *p = s;
    char *q = s;
    while(*q != '\0')
    {
        if (*q == ' ')
        {
            ReverseWord(p, q - 1);
            q++;
            p = q;
        }
        else
            q++;
    }
    ReverseWord(p, q - 1);
    ReverseWord(s, q - 1);
    return s;
}

int main( )
{
    char a[]="I am glad to see you";
    int len = sizeof(a)/sizeof(a[0]);
    printf("%s\n",Reverse(a));
    return 0;
}
```

程序输出结果:

you see to glad am I

引申: 如何实现逆序打印?

与上题类似,还有一类题目要求逆序输出,而不要求真正地逆序存储。对于这个问题,可以首先求出字符串长度,然后反向遍历即可。程序示例如下:


```

#include <stdio.h>
#include <string.h>

void ReversePrint(const char* s)
{
    int len = strlen(s);
    for (int i = len - 1; i >= 0; --i)
        printf("%c", s[i]);
}

int main( )
{
    char a[]="abcd";
    ReversePrint(a);
    printf("\n");
    return 0;
}

```

程序输出结果:

dcba

如果不想求字符串的长度,可以先遍历到末尾,然后再遍历回来,这要借助字符串的结束符'\0'。程序代码示例如下:

```

#include <stdio.h>

void ReversePrint(const char* s)
{
    const char* p = s;
    while (*p)
        *p++;
    --p;
    while (p >= s)
    {
        printf("%c", *p);
        --p;
    }
}

int main( )
{
    char a[]="abcd";
    ReversePrint(a);
    printf("\n");
    return 0;
}

```

对于上述方法,也可以使用递归的方式完成。程序示例代码如下:

```

void ReversePrint(const char* s)
{
    if(*(s + 1) != '\0')
        ReversePrint(s + 1);
    printf("%c", *s);
}

```

13.3.3 如何找出一个字符串中第一个只出现一次的字符

如何找出一个字符串中第一个只出现一次的字符?例如,输入 abac,则输出 b。本题可以采用 hash 法来实现。首先申请一个长度为 256 的表,对每个字符 hash 计数即可。因为 C/C++

中的 char 有 3 种类型: char、signed char 和 unsigned char。char 类型的符号是由编译器指定的,一般是有符号的,在对字符进行 hash 时,应该先将字符转为无符号类型;否则当下标为负值时,就会出现越界访问。

另外,可以用一个 buffer 数组记录当前找到的只出现一次的字符,避免对原字符串进行第二次遍历。

程序示例如下:

```
#include <stdio.h>

char GetChar(char str[])
{
    if (str == NULL)
        return 0;
    const int size = 256;
    unsigned count[size] = {0};
    char buffer[size];
    char *q = buffer;
    for (const char* p = str; *p != 0; ++p)
        if (++count[(unsigned char)*p] == 1)
            *q++ = *p;
    for (p = buffer; p < q; ++p)
        if (count[(unsigned char)*p] == 1)
            return *p;
    return 0;
}

int main( )
{
    printf("%c\n",GetChar("abac"));
    return 0;
}
```

程序输出结果:

b

13.3.4 如何输出字符串的所有组合

如何输出字符串的所有组合?例如,“abc”输出 a、b、c、ab、ac、bc、abc,假设字符串中的所有字符都不重复。

根据题意,如果字符串中有 n 个字符,那么一共需要输出 $2^n - 1$ 种组合。

最容易想到的方式是递归法,遍历字符串,每个字符只能取或不取。取该字符的话,就把该字符放到结果字符串中,遍历完毕后,输出结果字符串。

程序示例如下:

```
#include <stdio.h>
#include <string.h>

void CombineRecursiveImpl(const char* str, char* begin, char* end)
{
    if (*str == 0)
    {
        *end = 0;
        if (begin != end)
            printf("%s ",begin);
        return;
    }
```

```

    }
    CombineRecursiveImpl(str + 1, begin, end);
    *end = *str;
    CombineRecursiveImpl(str + 1, begin, end + 1);
}

void CombineRecursive(const char str[])
{
    if (str == NULL)
        return;
    const int MAXLENGTH = 64;
    char result[MAXLENGTH];
    CombineRecursiveImpl(str, result, result);
}

int main( )
{
    CombineRecursive("abc");
    printf("\n");
    return 0;
}

```

程序输出结果：

c b bc a ac ab abc

采用递归法求解，当 n 比较大时，效率很差，因为栈调用次数约为 2^n ，尾递归优化后也有 2^{n-1} 。为了提高效率，考虑到本题的特性，可以构造一个长度为 n 的 01 字符串（或二进制数）表示输出结果中最否包含某个字符。例如，“001”表示输出结果中不含字符 a、b，只含 c，即输出结果为 c，而“101”表示输出结果为 ac。原题就是要求输出“001”~“111”这 2^n-1 个组合对应的字符串。

程序示例代码如下：

```

#include <stdio.h>
#include <string.h>

void Combine(const char str[])
{
    if (str == NULL || *str == 0)
        return;
    const int MAXLENGTH = 64;
    int len = strlen(str);
    bool used[MAXLENGTH] = {0};
    char cache[MAXLENGTH];
    char *result = cache + len;
    *result = 0;
    while (1)
    {
        int index = 0;
        while (used[index])
        {
            used[index] = false;
            ++result;
            if (++index == len)
                return;
        }
        used[index] = true;
        *--result = str[index];
    }
}

```

```

        printf("%s ",result);
    }
}

```

```

int main( )
{
    Combine("abc");
    printf("\n");
    return 0;
}

```

程序输出结果

a b ab c ac bc abc

13.3.5 如何检查字符是否是整数？如果是，返回其整数值

在解答这个问题之前，先来看一个“函数”。int isdigit(char c) 函数用来判断字符 c 是否为数字，当 c 为数字 0~9 时，返回非零值；否则返回零。使用时需要包含头文件<ctype.h>。此“函数”为宏定义，而非真正函数。另外，'0'的 ASCII 码为 48，因此如果一个字符为数字，那它减去 48 之后，存成整型类型，即可获得这个数的值。根据上面两点可以实现本题，程序示例如下：

```

#include <stdio.h>
#include <ctype.h>

```

```

int cheak( int p )
{
    int c=p;
    if (isdigit(p))
        c = p-48;
    return c;
}

```

```

int main( )
{
    int c;
    while ( ( c=getchar( ))!=EOF )
    {
        getchar( );
        c = cheak(c);
        if (isalpha(c))
            printf("不是\n");
        else
            printf("是: %d\n",c);
    }
    return 0;
}

```

程序输出结果：

```

1
是: 1
a
不是

```

13.3.6 如何查找字符串中每个字符出现的个数

写出一个函数，查找出每个字符的个数，主要区分大小写，要求时间复杂度是 $O(n)$ 。

用 256 个元素的数组 `count`，来分别记录 ASCII 码为 0~255 的字符的个数，初始化为 0，遍历每个字符，对该字符对应的数组元素的值加 1。最后 `count[i]` 中存储的数值就为字符 `i` 的个数。具体实现如下：

```
#include<stdio.h>
int main( )
{
    char *str="AbcABca";
    int count[256]={0};
    for(char *p=str;*p;p++)
    {
        count[*p]++;
    }
    for(int i=0;i<256;i++)
    {
        if(count[i]>0)
        {
            printf("The count of %c is: %d\n",i,count[i]);
        }
    }
    return 0;
}
```

程序的输出结果：

```
The count of A is : 2
The count of B is : 1
The count of a is : 1
The count of b is : 1
The count of c is : 2
```

13.4 STL 容器

STL (Standard Template Library) 是一个 C++ 领域中，用模板技术实现的数据结构和算法库，其中的 `vector`、`list`、`stack`、`queue` 等结构不仅拥有更强大的功能，还有了更高的安全性。它体现了泛型编程的思想，具有高度的可重用性、高性能、高移植性。程序员不用思考具体的实现过程，只要能够熟练应用即可。

13.4.1 什么是泛型编程

泛型编程 (Generic Programming) 的目的是为了发明一种语言机制，能够帮助实现一个通用的标准容器库。通用的标准容器库是指能够实现这样一种功能：例如，用一个 `List` 类存放所有可能类型的对象，而泛型编程可以让程序员编写完全一般化并可重复使用的算法，其效率与针对某特定数据类型而设计的算法相同。泛型与模板类似，指具有在多种数据类型上皆可操作的含意。

STL 巨大，而且可以扩充，它包含很多计算机基本算法和数据结构，而且将算法与数据结构完全分离，其中算法是泛型的，不与任何特定数据结构或对象类型系在一起。

13.4.2 栈与队列的区别有哪些

栈与队列是在程序设计中被广泛使用的两种重要的线性数据结构，都是在一个特定范围的存储单元中存储的数据，这些数据都可以重新被取出使用，与线性表相比，它们的插入和删除

操作受更多的约束和限定, 故又称为限定性的线性表结构。不同的是, 栈就像一个很窄的桶先存进去的数据只能最后才能取出来, 是 LIFO (Last In First Out, 后进先出), 它将进出顺序逆序, 即先进后出, 后进先出。栈结构如图 13-13 所示。队列像日常排队买东西的人的“队列”, 先排队的人先买, 后排队的人后买, 是 FIFO (First In First Out, 先进先出) 的, 它保持进出顺序一致, 即先进先出, 后进后出。队列结构如图 13-14 所示。

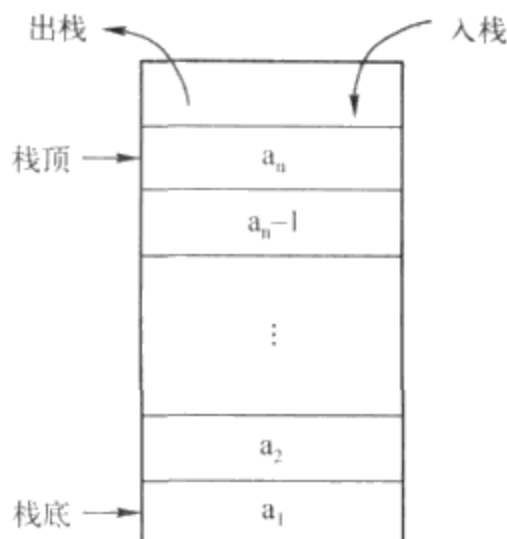


图 13-13 栈结构示意图

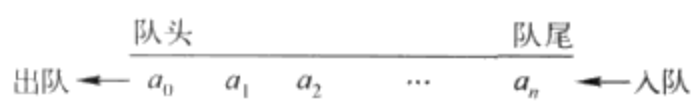


图 13-14 队列结构示意图

需要注意的是, 有时在数据结构中还有可能出现按照大小排队或按照一定条件排队的数据队列, 这时的队列属于特殊队列, 就不一定按照“先进先出”的原则读取数据了。

13.4.3 vector 与 list 的区别有哪些

vector 为存储的对象分配一块连续的地址空间, 对 vector 中的元素随机访问效率很高。在 vector 中插入或者删除某个元素, 需要将现有元素进行复制、移动。如果 vector 中存储的对象很大, 或者构造函数复杂, 则在对现有元素进行拷贝时开销较大, 因为拷贝对象要调用拷贝构造函数。对于简单的小对象, vector 的效率优于 list。vector 在每次扩张容量的时候, 将容量扩展 2 倍, 这样对于小对象来说, 效率是很高的。

list 表示非连续的内存区域, 并通过一对指向首尾元素的指针双向链接起来从而允许向前和向后两个方向进行遍历, list 中的对象是离散存储的。在 list 的任意位置插入与删除元素的效率都很高, 指针必须被重新赋值, 但是不需要用拷贝元素来实现移动。它对随机访问的支持并不好, 访问一个元素需要遍历中间的元素, 另外每个元素还有两个指针的额外空间开销, 随机访问某个元素需要遍历 list。在 list 中插入元素, 尤其是在首尾插入元素, 效率很高, 只需要改变元素的指针即可。

vector 内部使用顺序存储, 访问速度快, 但是删除数据比较耗费性能。List 内部使用链式存储, 访问速度慢, 但是删除数据比较快。

一般应遵循下面的原则:

- (1) 需要高效的随机存取, 而不在于插入和删除的效率, 使用 vector。
- (2) 需要大量的插入和删除, 而不关心随机存取, 则应使用 list。
- (3) 需要随机存取, 而且关心两端数据的插入和删除, 则应使用 deque。

13.4.4 如何实现循环队列

在队列的顺序存储结构中, 除了使用一组地址连续的存储单元依次存放从队列头到队列尾的元素之外, 还需要另外设置两个指针 front 和 rear 分别指示队列头元素以及队列尾元素的位置。

置。初始化建空队列时，令 $\text{front}=\text{rear}=0$ ，每当插入新的队列尾元素时，“尾指针增 1”，而每当删除队列头元素时，则执行“头指针增 1”。因此，在非空队列中，头指针始终指向队列头元素，尾指针始终指向队列尾元素的下一个位置。

为充分利用向量空间，克服“假溢出”现象的方法是：将向量空间想象为一个首尾相接的圆环，并称这种向量为循环向量。存储在其中的队列称为循环队列（Circular Queue）。循环队列中，由于入队时尾指针向前追赶头指针；出队时头指针向前追赶尾指针，造成队空和队满时头尾指针均相等。因此，无法通过条件 $\text{front}=\text{rear}$ 来判别队列是“空”还是“满”。

解决这个问题的方法至少有两种：

(1) 另外设置一个标志位来区别队列是“空”还是“满”。

(2) 少用一个元素空间，约定以“队列头指针在队列尾指针的下一位置（指环状的下一位置）”上作为队列呈“满”状态的标志。队满时： $(\text{rear}+1)\%n=\text{front}$ ， n 为队列长度（所用数组大小），由于 rear 、 front 均为所用空间的指针，循环只是逻辑上的循环，所以需要求余运算。

算法示例如下：

```
#define MAXSIZE 1000
typedef int ElemType;
typedef struct
{
    ElemType data[MAXSIZE];
    int front;
    int rear;
}CircSeqQueue;

//顺序循环队列的初始化
void QueueInitial(CircSeqQueue *pQ)
{
    pQ->front=pQ->rear=0;
}

//顺序循环队列判空
int IsEmpty(CircSeqQueue *pQ)
{
    return pQ->front==pQ->rear;
}

//顺序循环队列判满
int IsFull(CircSeqQueue *pQ)
{
    return (pQ->rear+1)%MAXSIZE==pQ->front;
}

//元素进队列
void EnQueue(CircSeqQueue *pQ, ElemType e)
{
    if(IsFull(pQ))
    {
        printf("队列溢出！\n");
        exit(1);
    }
    pQ->rear=(pQ->rear+1)%MAXSIZE;
    pQ->data[pQ->rear]=e;
}
```

```

//元素出队列
ElemType DeQueue(CircSeqQueue *pQ)
{
    if(IsEmpty(pQ))
    {
        printf("空队列!\n");
        exit(1);
    }
    pQ->front=(pQ->front+1)%MAXSIZE;
    return pQ->data[pQ->front];
}

//取队头元素
ElemType GetFront(CircSeqQueue *pQ)
{
    if(IsEmpty(pQ))
    {
        printf("队列为空!\n");
        exit(1);
    }
    return pQ->data[(pQ->front+1)%MAXSIZE];
}

//循环队列置空
void MakeEmpty(CircSeqQueue *pQ)
{
    pQ->front=pQ->rear=0;
}

```

13.4.5 如何使用两个栈模拟队列操作

题目要求用两个栈来模拟队列，栈 A 与栈 B 模拟队列 Q，A 为插入栈，B 为弹出栈，以实现队列 Q。

假设 A 和 B 都为空，可以认为栈 A 提供入队列的功能，栈 B 提供出队列的功能。

入队列：入栈 A。

出队列分两种情况考虑：

- (1) 如果栈 B 不为空，则直接弹出栈 B 的数据。
- (2) 如果栈 B 为空，则依次弹出栈 A 的数据，放入栈 B 中，再弹出栈 B 的数据。

```

#include <iostream>
#include <stack>
using namespace std;

template <typename T>
class QueueByDoubleStack
{
public:
    size_t size( );
    bool empty( );
    void push(T t);
    void pop( );
    T top( );
private:
    stack<T> s1;
    stack<T> s2;
}

```

```
};

template <typename T>
size_t QueueByDoubleStack<T>::size( )
{
    return s1.size( ) + s2.size( );
}

template <typename T>
bool QueueByDoubleStack<T>::empty( )
{
    return s1.empty( ) && s2.empty( );
}

template <typename T>
void QueueByDoubleStack<T>::push(T t)
{
    s1.push(t);
}

template <typename T>
void QueueByDoubleStack<T>::pop( )
{
    if (s2.empty( ))
    {
        while (!s1.empty( ))
        {
            s2.push(s1.top( ));
            s1.pop( );
        }
    }
    s2.pop( );
}

template <typename T>
T QueueByDoubleStack<T>::top( )
{
    if (s2.empty( ))
    {
        while (!s1.empty( ))
        {
            s2.push(s1.top( ));
            s1.pop( );
        }
    }
    return s2.top( );
}

int main( )
{
    QueueByDoubleStack<int> q;
    for (int i = 0; i < 10; ++i)
    {
        q.push(i);
    }
    while (!q.empty( ))
    {

```

```

        cout << q.top() << ' ';
        q.pop();
    }
    cout << endl;
    return 0;
}

```

程序输出结果:

0 1 2 3 4 5 6 7 8 9

引申: 如何使用两个队列实现栈?

可以采用两种方法实现, 入栈: 所有元素依次入队列 q1。例如, 将 A、B、C、D 四个元素入栈, 从队列尾部到队列首部依次为 D、C、B、A, 出栈的时候判断栈元素个数是否为 1, 如果为 1, 则队列 q1 出列; 如果不为 1, 则队列 q1 所有元素出队列, 入队列 q2, 最后一个元素不入队列 B, 输出该元素, 队列 q2 所有元素入队列 q1。例如, 将 D、C、B、A 出列, D 输出, C、B、A 入队列 q2, 最后返回到队列 q1 中, 实现了后进先出。

13.5 排序

排序问题一直是计算机技术研究的重要问题, 排序算法的好坏直接影响程序的执行速度和辅助存储空间占有量, 所以各大 IT 企业在笔试面试中也经常出现有关排序的题目。本节将详细分析常见的各种排序算法, 并从时间复杂度、空间复杂度、适用情况等多个方面对它们进行综合比较。

13.5.1 如何进行选择排序

选择排序是一种简单直观的排序算法, 它的基本原理如下: 对于给定的一组记录, 经过第一轮比较后得到最小的记录, 然后将该记录与第一个记录的位置进行交换; 接着对不包括第一个记录以外的其他记录进行第二轮比较, 得到最小的记录并与第二个记录进行位置交换; 重复该过程, 直到进行比较的记录只有一个时为止。以数组 {38, 65, 97, 76, 13, 27, 49} 为例, 具体步骤如下:

第一趟排序后: 13 [65 97 76 38 27 49]

第二趟排序后: 13 27 [97 76 38 65 49]

第三趟排序后: 13 27 38 [76 97 65 49]

第四趟排序后: 13 27 38 49 [97 65 76]

第五趟排序后: 13 27 38 49 65 [97 76]

第六趟排序后: 13 27 38 49 65 76 [97]

最后排序结果: 13 27 38 49 65 76 97

程序示例如下:

```

#include<stdio.h>

void SelectSort(int *a,int n)
{
    int i;
    int j;
    int temp = 0;
    int flag = 0;
    for(i = 0; i < n-1; i++)

```



```

    {
        temp = a[i];
        flag = i;
        for(j = i+1; j < n; j++)
        {
            if(a[j] < temp)
            {
                temp = a[j];
                flag = j;
            }
        }
        if(flag != i)
        {
            a[flag] = a[i];
            a[i] = temp;
        }
    }
}

int main( )
{
    int i = 0;
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    int len = sizeof(a)/sizeof(a[0]);
    SelectSort(a, len);
    for(i = 0; i < len; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}

```

程序输出结果:

0 1 2 3 4 5 6 7 8 9

从简单选择排序的过程来看,它的特点就是交换移动数据次数相当少,这样也就节约了相应的时间。无论是最好情况,还是最差情况,其比较次数都是一样的,第 i 趟排序需要进行 $n-i$ 次。而对于交换次数而言,最好的情况是有序,需要交换 0 次;最差的情况,即逆序时,交换次数为 $n-1$ 次,基于最终的排序时间是比较与交换的次数总和,因此总的时间复杂度依然为 $O(n^2)$ 。

13.5.2 如何进行插入排序

对于给定的一组记录,初始时假设第一个记录自成一个有序序列,其余的记录为无序序列;接着从第二个记录开始,按照记录的大小依次将当前处理的记录插入到其之前的有序序列中,直至最后一个记录插入到有序序列中为止。以数组 {38, 65, 97, 76, 13, 27, 49} 为例,直接插入排序具体步骤如下:

第一步插入 38 以后: [38] 65 97 76 13 27 49

第二步插入 65 以后: [38 65] 97 76 13 27 49

第三步插入 97 以后: [38 65 97] 76 13 27 49

第四步插入 76 以后: [38 65 76 97] 13 27 49

第五步插入 13 以后: [13 38 65 76 97] 27 49

第六步插入 27 以后: [13 27 38 65 76 97] 49

第七步插入 49 以后: [13 27 38 49 65 76 97]

程序示例如下:

```
#include <stdio.h>

void InsertSort(int par_array[], int array_size)
{
    int i,j;
    int temp;
    for(i = 1; i < array_size; i++)
    {
        temp = par_array[i];
        for(j = i-1; j >= 0; j--)
        {
            if(temp < par_array[j])
            {
                par_array[j+1] = par_array[j];
            }
            else
                break;
        }
        par_array[j+1] = temp;
    }
}

int main( )
{
    int i = 0;
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    int len = sizeof(a)/sizeof(a[0]);
    InsertSort(a, len);
    for(i = 0; i < len; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

程序输出结果:

0 1 2 3 4 5 6 7 8 9

13.5.3 如何进行冒泡排序

冒泡排序顾名思义就是整个过程就像气泡一样往上升, 单向冒泡排序的基本思想是(假设由小到大排序): 对于给定的 n 个记录, 从第一个记录开始依次对相邻的两个记录进行比较, 当前面的记录大于后面的记录时, 交换其位置, 进行一轮比较和换位后, n 个记录中的最大记录将位于第 n 位; 然后对前 $(n-1)$ 个记录进行第二轮比较; 重复该过程直到进行比较的记录只剩下一个时为止。

以数组 {36, 25, 48, 12, 25, 65, 43, 57} 为例, 具体排序过程如下:

一趟排序的过程如下:

| | | | | | | | | |
|------|----|----|----|----|----|----|----|----|
| R[1] | 36 | 25 | 25 | 25 | 25 | 25 | 25 | 25 |
| R[2] | 25 | 36 | 36 | 36 | 36 | 36 | 36 | 36 |
| R[3] | 48 | 48 | 48 | 12 | 12 | 12 | 12 | 12 |
| R[4] | 12 | 12 | 12 | 48 | 25 | 25 | 25 | 25 |
| R[5] | 25 | 25 | 25 | 25 | 48 | 48 | 48 | 48 |

R[6] 65 65 65 65 65 65 43 43
 R[7] 43 43 43 43 43 43 65 57
 R[8] 57 57 57 57 57 57 57 65

则经过多趟排序后的结果如下:

初始状态: [36 25 48 12 25 65 43 57]

1 趟排序: [25 36 12 25 48 43 57 65]

2 趟排序: [25 12 25 36 43 48] 57 65

3 趟排序: [12 25 25 36 43] 48 57 65

4 趟排序: [12 25 25 36] 43 48 57 65

5 趟排序: [12 25 25] 36 43 48 57 65

6 趟排序: [12 25] 25 36 43 48 57 65

7 趟排序: [12] 25 25 36 43 48 57 65

程序示例如下:

```
#include <stdio.h>

void Swap(int& a,int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void BubbleSort(int array[], int len)
{
    int i,j;
    for (i = 0; i < len-1; ++i)
    {
        for (j = len-1; j > i; --j)
        {
            if (array[j] < array[j-1])
            {
                Swap(array[j],array[j-1]);
            }
        }
    }
}

int main( )
{
    int i = 0;
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    int len = sizeof(a)/sizeof(a[0]);
    BubbleSort(a, len);
    for(i = 0; i < len; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

程序输出结果:

0 1 2 3 4 5 6 7 8 9

引申: 如何进行双向冒泡排序?

双向冒泡排序是冒泡排序的一种优化, 它的基本思想是首先将第一个记录的关键字和第二

个记录的关键字进行比较,若为“逆序”(即 $L.r[1].key > L.r[2].key$),则将两个记录交换,然后比较第二个记录和第三个记录的关键字。依次类推,直至第 $n-1$ 个记录的关键字和第 n 个记录的关键字比较过为止。这是第一趟冒泡排序,其结果是使得关键字最大的记录被安置到最后一个记录的位置上。

第一趟排序之后进行第二趟冒泡排序,将第 $n-2$ 个记录的关键字和第 $n-1$ 个记录的关键字进行比较,若为“逆序”(即 $L.r[n-1].key < L.r[n-2].key$),则将两个记录交换,然后比较第 $n-3$ 个记录和 $n-2$ 个记录的关键字。依次类推,直至第 1 个记录的关键字和第 2 个记录的关键字比较过为止。其结果是使得关键字最小的记录被安置到第一个位置上。

再对其余的 $n-2$ 个记录进行上述同样的操作,其结果是使关键字次大的记录被安置到第 $n-1$ 个记录的位置,使关键字次小的记录被安置到第 2 个记录的位置。

一般地,第 i 趟冒泡排序是:若 i 为奇数,则从 $L.r[i/2+1] \sim L.r[n-i/2]$ 依次比较相邻两个记录的关键字,并在“逆序”时交换相邻记录,其结果是这 $n-i+1$ 个记录中关键字最大的记录被交换到第 $n-i/2$ 的位置上;若 i 为偶数,则从 $L.r[n-i/2] \sim L.r[i/2]$ 依次比较相邻两个记录的关键字,并在“逆序”时交换相邻记录,其结果是这 $n-i+1$ 个记录中关键字最小的记录被交换到第 $i/2$ 的位置上。整个排序过程需要进行 K ($1 \leq K < n$) 趟冒泡排序,同样判别冒泡排序结束的条件仍然是“在一趟排序过程中没有进行过交换记录的操作”。

程序示例如下:

```
#include <stdio.h>
void Swap(int& a,int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void Bubble2Sort(int array[],int length)
{
    int left = 1;
    int right = length - 1;
    int t;
    do
    {
        //正向的部分
        for(int i=right;i>=left;i--)
        {
            if(array[i]<array[i-1])
            {
                Swap(array[i],array[i-1]);
                t = i;
            }
        }
        left = t+1;
        //反向的部分
        for(i=left;i<right+1;i++)
        {
            if(array[i]<array[i-1])
            {
                Swap(array[i],array[i-1]);
```

```

        t = i;
    }
}
right = t-1;
}while(left<=right);
}

int main( )
{
    int i = 0;
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    int len = sizeof(a)/sizeof(a[0]);
    Bubble2Sort(a, len);
    for(i = 0; i < len; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}

```

程序输出结果:

0 1 2 3 4 5 6 7 8 9

13.5.4 如何进行归并排序

归并排序是利用递归与分治技术将数据序列划分成为越来越小的半子表，再对半子表排序，最后再用递归步骤将排好序的半子表合并成为越来越大的有序序列。其中“归”代表的是递归的意思，即递归地将数组折半地分离为单个数组。例如，数组[2, 6, 1, 0]会先折半，分为[2, 6]和[1, 0]两个子数组，然后再折半将数组分离，分为[2]，[6]和[1]，[0]。“并”就是将分开的数据按照从小到大或者从大到小的顺序在放到一个数组中。如上面的[2]、[6]合并到一个数组中是[2, 6]，[1]、[0]合并到一个数组中是[0, 1]，然后再将[2, 6]和[0, 1]合并到一个数组中即为[0, 1, 2, 6]。

具体而言，归并排序算法的原理如下：对于给定的一组记录（假设共有 n 个记录），首先将每两个相邻的长度为 1 的子序列进行归并，得到 $n/2$ （向上取整）个长度为 2 或 1 的有序子序列，再将其两两归并，反复执行此过程，直到得到一个有序序列为止。

所以，归并排序的关键就是两步：第一步，划分子表；第二步，合并半子表。以数组{49, 38, 65, 97, 76, 13, 27}为例，排序过程如下：

```

初始关键字: [49] [38] [65] [97] [76] [13] [27]
              └──┘ └──┘ └──┘
一趟归并后: [38 49] [65 97] [13 76] [27]
              └──┘ └──┘
二趟归并后: [38 49 65 97] [13 27 76]
              └──┘
三趟归并后: [13 27 38 49 65 76 97]

```

程序示例如下：

```

#include <stdio.h>

void Merge(int array[], int p, int q, int r)
{
    int i, j, k, n1, n2;
    n1 = q - p + 1;
    n2 = r - q;

```



```

int* L = new int[n1];
int* R = new int[n2];
for(i = 0, k = p; i < n1; i++, k++)
    L[i] = array[k];
for(i = 0, k = q + 1; i < n2; i++, k++)
    R[i] = array[k];
for(k = p, i = 0, j = 0; i < n1 && j < n2; k++)
{
    if(L[i] > R[j])
    {
        array[k] = L[i];
        i++;
    }
    else
    {
        array[k] = R[j];
        j++;
    }
}
if(i < n1)
{
    for(j = i; j < n1; j++, k++)
        array[k] = L[j];
}
if(j < n2)
{
    for(i = j; i < n2; i++, k++)
        array[k] = R[i];
}
}

```

```

void MergeSort(int array[], int p, int r)
{
    if(p < r)
    {
        int q = (p + r) / 2;
        MergeSort(array, p, q);
        MergeSort(array, q + 1, r);
        Merge(array, p, q, r);
    }
}

```

```

int main( )
{
    int i = 0;
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    int len = sizeof(a)/sizeof(a[0]);
    MergeSort(a,0,len-1);
    for(i = 0; i < len; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}

```

程序输出结果:

9 8 7 6 5 4 3 2 1 0

二路归并排序的过程需要进行 $\log n$ 趟。每一趟归并排序的操作,就是将两个有序子序列进行归并,而每一对有序子序列归并时,记录的比较次数均小于等于记录的移动次数,记录移

动的次数均等于文件中记录的个数 n ，即每一趟归并的时间复杂度为 $O(n)$ 。因此，二路归并排序的时间复杂度为 $O(n\log n)$ 。

13.5.5 如何进行快速排序

快速排序是一种非常高效的排序算法，它采用“分而治之”的思想，把大的拆分为小的，小的再拆分为更小的。其原理是：对于一组给定的记录，通过一趟排序后，将原序列分为两部分，其中前部分的所有记录均比后部分的所有记录小，然后再依次对前后两部分的记录进行快速排序，递归该过程，直到序列中的所有记录均有序为止。

具体算法步骤如下。

(1) 分解：将输入的序列 $\text{array}[m, \dots, n]$ 划分成两个非空子序列 $\text{array}[m, \dots, k]$ 和 $\text{array}[k+1, \dots, n]$ ，使 $\text{array}[m, \dots, k]$ 中任一元素的值不大于 $\text{array}[k+1, \dots, n]$ 中任一元素的值。

(2) 递归求解：通过递归调用快速排序算法分别对 $\text{array}[m, \dots, k]$ 和 $\text{array}[k+1, \dots, n]$ 进行排序。

(3) 合并：由于对分解出的两个子序列的排序是就地进行的，所以在 $\text{array}[m, \dots, k]$ 和 $\text{array}[k+1, \dots, n]$ 都排好序后，不需要执行任何计算 $\text{array}[m, \dots, n]$ 就已排好序。

以数组 $\{38, 65, 97, 76, 13, 27, 49\}$ 为例。

第一趟排序过程如下：

初始化关键字 $[49\ 38\ 65\ 97\ 76\ 13\ 27\ 49]$

第一次交换后： $[27\ 38\ 65\ 97\ 76\ 13\ 49\ 49]$

第二次交换后： $[27\ 38\ 49\ 97\ 76\ 13\ 65\ 49]$

j 向左扫描，位置不变，第三次交换后： $[27\ 38\ 13\ 97\ 76\ 49\ 65\ 49]$

i 向右扫描，位置不变，第四次交换后： $[27\ 38\ 13\ 49\ 76\ 97\ 65\ 49]$

j 向左扫描 $[27\ 38\ 13\ 49\ 76\ 97\ 65\ 49]$

整个排序过程如下：

初始化关键字 $[49\ 38\ 65\ 97\ 76\ 13\ 27\ 49]$

一趟排序之后： $[27\ 38\ 13]\ 49\ [76\ 97\ 65\ 49]$

二趟排序之后： $[13]\ 27\ [38]\ 49\ [49\ 65]\ 76\ [97]$

三趟排序之后： $13\ 27\ 38\ 49\ 49\ [65]\ 76\ 97$

最后的排序结果： $13\ 27\ 38\ 49\ 49\ 65\ 76\ 97$

程序示例如下：

```
#include <stdio.h>

void Sort(int array[], int low, int high)
{
    int i, j;
    int index;
    if(low >= high)
        return ;
    i = low;
    j = high;
    index = array[i];
    while (i < j)
    {
        while (i < j && array[j] >= index)
            j--;
        if(i < j)
```

```

        array[i++] = array[j];
        while (i < j && array[i] < index)
            i++;
        if(i < j)
            array[j--] = array[i];
    }
    array[i] = index;
    Sort(array, low, i-1);
    Sort(array, i+1, high);
}

```

```

void QuickSort(int array[], int len)
{
    Sort(array, 0, len-1);
}

```

```

int main( )
{
    int i = 0;
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    int len = sizeof(a)/sizeof(a[0]);
    QuickSort(a, len);
    for(i = 0; i < len; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}

```

程序输出结果:

0 1 2 3 4 5 6 7 8 9

当初始的序列整体或局部有序时,快速排序的性能将会下降,此时快速排序将退化为冒泡排序。

快速排序的相关特点如下:

(1) 最坏时间复杂度。

最坏情况是指每次区间划分的结果都是基准关键字的左边(或右边)序列为空,而另一边的区间中的记录项仅比排序前少了一项,即选择的基准关键字是待排序的所有记录中最小或者最大的。例如,若选取第一个记录为基准关键字,当初始序列按递增顺序排列时,每次选择的基准关键字都是所有记录中的最小者,这时记录与基准关键字的比较次数会增多。因此,在这种情况下,需要进行 $(n-1)$ 次区间划分。对于第 k ($0 < k < n$)次区间划分,划分前的序列长度为 $(n-k+1)$,需要进行 $(n-k)$ 次记录的比较。当 k 从 $1 \sim (n-1)$ 时,进行的比较次数总共为 $n(n-1)/2$,所以在最坏情况下快速排序的时间复杂度为 $O(n^2)$ 。

(2) 最好时间复杂度。

最好情况是指每次区间划分的结果都是基准关键字左右两边的序列长度相等或者相差为1,即选择的基准关键字为待排序的记录中的中间值。此时,进行的比较次数总共为 $n \log n$,所以在最好情况下快速排序的时间复杂度为 $O(n \log n)$ 。

(3) 平均时间复杂度。

快速排序的平均时间复杂度为 $O(n \log n)$ 。虽然快速排序在最坏情况下的时间复杂度为 $O(n^2)$,但是在所有平均时间复杂度为 $O(n \log n)$ 的算法中,快速排序的平均性能是最好的。

(4) 空间复杂度。

快速排序的过程中需要一个栈空间来实现递归。当每次对区间的划分都比较均匀时(即最

好情况), 递归树的最大深度为 $\lceil \log n \rceil + 1$ ($\log n$ 为向上取整); 当每次区间划分都使得有一边的序列长度为 0 时 (即最好情况), 递归树的最大深度为 n 。在每轮排序结束后比较基准关键字左右的记录个数, 对记录多的一边先进行排序, 此时, 栈的最大深度可降为 $\log n$ 。因此, 快速排序的平均空间复杂度为 $O(\log n)$ 。

(5) 基准关键字的选取。

基准关键字的选择是决定快速排序算法性能的关键。常用的基准关键字的选择有以下几种方式:

1) 三者取中。

三者取中是指在当前序列中, 将其首、尾和中间位置上的记录进行比较, 选择三者的中值作为基准关键字, 在划分开始前交换序列中的第一个记录与基准关键字的位置。

2) 取随机数。

取 $left$ (左边) 和 $right$ (右边) 之间的一个随机数 $m(left \leq m \leq right)$, 用 $n[m]$ 作为基准关键字。这种方法使得 $n[left] \sim n[right]$ 之间的记录是随机分布的, 采用此方法得到的快速排序一般称为随机的快速排序。

需要注意快速排序与归并排序的区别与联系。快速排序与归并排序的原理都是基于分治思想, 即首先把待排序的元素分成两组, 然后分别对这两组排序, 最后把两组结果合并起来。

而它们的不同点在于, 进行的分组策略不同, 后面的合并策略也不同。归并排序的分组策略是假设待排序的元素存放在数组中, 那么其把数组前面一半元素作为一组, 后面一半元素作为另外一组。而快速排序则是根据元素的值来分组, 即大于某个值的元素放在一组, 而小于的放在另外一组, 该值称为基准。所以, 对整个排序过程而言, 基准值的挑选非常重要, 如果选择不合适, 太大或太小, 那么所有的元素都分在一组了。对于快速排序和归并排序来说, 如果分组策略越简单, 则后面的合并策略就越复杂, 因为快速排序在分组时, 已经根据元素大小来分组了, 而合并的时候, 只需把两个分组合并起来就行了, 归并排序则需要对两个有序的数组根据大小合并。

13.5.6 如何进行希尔排序

希尔排序也称为“缩小增量排序”。它的基本原理是: 首先将待排序的元素分成多个子序列, 使得每个子序列的元素个数相对较少, 对各个子序列分别进行直接插入排序, 待整个待排序序列“基本有序后”, 再对所有元素进行一次直接插入排序。

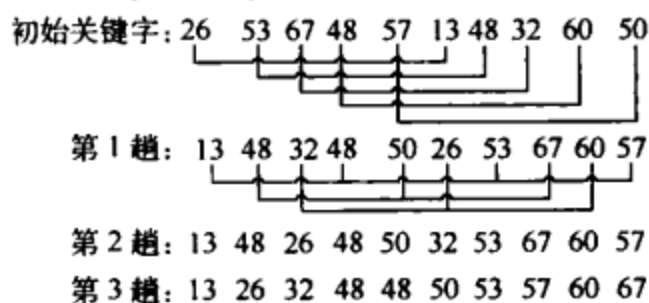
具体步骤如下:

(1) 选择一个步长序列 t_1, t_2, \dots, t_k , 满足 $t_i > t_j (i < j)$, $t_k = 1$ 。

(2) 按步长序列个数 k , 对待排序序列进行 k 趟排序。

(3) 每趟排序, 根据对应的步长 t_i , 将待排序列分割成 t_i 个子序列, 分别对各个子序列进行直接插入排序。

注意当步长因子为 1 时, 所有元素作为一个序列来处理, 其长度为 n 。以数组 {26, 53, 67, 48, 57, 13, 48, 32, 60, 50}, 步长序列 {5, 3, 1} 为例。具体步骤如下:



程序示例如下:

```
#include <stdio.h>

void ShellSort(int array[], int length)
{
    int i, j;
    int h;
    int temp;
    for(h = length/2; h > 0; h=h/2)
    {
        for(i = h; i < length; i++)
        {
            temp = array[i];
            for(j = i-h; j >= 0; j-=h)
            {
                if(temp < array[j])
                {
                    array[j+h] = array[j];
                }
                else
                    break;
            }
            array[j+h] = temp;
        }
    }
}

int main( )
{
    int i = 0;
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    int len = sizeof(a)/sizeof(a[0]);
    ShellSort(a, len);
    for(i = 0; i < len; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

程序输出结果:

0 1 2 3 4 5 6 7 8 9

希尔排序的关键并不是随便地分组后各自排序,而是将相隔某个“增量”的记录组成一个子序列,实现跳跃式的移动,使得排序的效率提高。

13.5.7 如何进行堆排序

堆是一种特殊的树形数据结构,其每个结点都有一个值,通常提到的堆都是指一棵完全二叉树,根结点的值小于(或大于)两个子结点的值,同时根结点的两个子树也分别是一个堆。

堆排序是一树形选择排序,在排序过程中,将 $R[1, \dots, N]$ 看成是一棵完全二叉树的顺序存储结构,利用完全二叉树中双亲结点和孩子结点之间的内在关系来选择最小的元素。

堆一般分为大顶堆和小顶堆两种不同的类型。对于给定 n 个记录的序列 $(r(1), r(2), \dots, r(n))$, 当且仅当满足条件 $(r(i) \geq r(2i), i=1, 2, \dots, n)$ 时称之为大顶堆,此时堆顶元素必为最大值。对于给定 n 个记录的序列 $(r(1), r(2), \dots, r(n))$, 当且仅当满足条件 $(r(i) \leq r(2i+1), i=1, 2, \dots, n)$ 时称之为小顶

堆，此时堆顶元素必为最小值。

堆排序的思想是对于给定的 n 个记录，初始时把这些记录看做一棵顺序存储的二叉树，然后将其调整为一个最大堆，然后将堆的最后一个元素与堆顶元素（即二叉树的根结点）进行交换后，堆的最后一个元素即为最大记录；接着将前 $(n-1)$ 个元素（即不包括最大记录）重新调整为一个最大堆，再将堆顶元素与当前堆的最后一个元素进行交换后得到次大的记录，重复该过程直到调整的堆中只剩一个元素时为止，该元素即为最小记录，此时可得到一个有序序列。

堆排序主要包括两个过程：一是构建堆；二是交换堆顶元素与最后一个元素的位置。

程序示例如下：

```
#include <stdio.h>

void AdjustMinHeap(int *a, int pos, int len)
{
    int temp;
    int child;
    for (temp = a[pos]; 2 * pos + 1 <= len; pos = child)
    {
        child = 2 * pos + 1;
        if (child < len && a[child] > a[child + 1])
            child++;
        if (a[child] < temp)
            a[pos] = a[child];
        else
            break;
    }
    a[pos] = temp;
}

void Swap(int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void MyMinHeapSort(int *array, int len)
{
    int i;
    for (i = len/2 - 1; i >= 0; i--)
        AdjustMinHeap(array, i, len - 1);
    for (i = len - 1; i >= 0; i--)
    {
        Swap(array[0], array[i]);
        AdjustMinHeap(array, 0, i - 1);
    }
}

void PrintArray(int *a, int length)
{
    int i;
    for (i = 0; i < length; i++)
        printf("%d ", a[i]);
    printf("\n");
}
```

```

int main( )
{
    int array[]={0,13,1,14,27,18};
    int length = sizeof(array)/sizeof(array[0]);
    MyMinHeapSort(array,length);
    PrintArray(array,length);
    return 0;
}

```

程序输出结果:

27 18 14 13 1 0

堆排序方法对记录较少的文件效果一般,但对于记录较多的文件还是很有效的,其运行时间主要耗费在创建堆和反复调整堆上。堆排序即使在最坏情况下,其时间复杂度也为 $O(N \times \log N)$ 。

13.5.8 各种排序算法有什么优劣

各种算法的性能见表 13-1。

表 13-1 排序算法比较

| 排序方法 | 最好时间 | 平均时间 | 最坏时间 | 辅助存储 | 稳定性 | 备注 |
|--------|---------------|---------------|----------------------|-------------|-----|-----------|
| 简单选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 | n 小时较好 |
| 直接插入排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 稳定 | 大部分已有序时较好 |
| 冒泡排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 稳定 | n 小时较好 |
| 希尔排序 | $O(n)$ | $O(n \log n)$ | $O(n^s) \ 1 < s < 2$ | $O(1)$ | 不稳定 | s 是所选分组 |
| 快速排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(\log n)$ | 不稳定 | n 大时较好 |
| 堆排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | 不稳定 | n 大时较好 |
| 归并排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | 稳定 | n 大时较好 |

从该表中可以得到以下几个方面的结论:

(1) 简单地说,所有相等的数经过某种排序方法后,仍能保持它们在排序之前的相对次序,就称这种排序方法是稳定的,反之就是非稳定的。例如,一组数排序前是 a_1, a_2, a_3, a_4, a_5 , 其中 $a_2=a_4$, 经过某种排序后为 a_1, a_2, a_4, a_3, a_5 , 则说这种排序是稳定的,因为 a_2 排序前在 a_4 的前面,排序后它还是在 a_4 的前面。假如变成 a_1, a_4, a_2, a_3, a_5 就不是稳定的了。各种排序算法中稳定的排序算法有直接插入排序、冒泡排序和归并排序,而不稳定的排序算法有希尔排序、快速排序、简单选择排序和堆排序。

(2) 时间复杂度为 $O(n^2)$ 的排序算法有直接插入排序、冒泡排序、快速排序和简单选择排序,时间复杂度为 $O(n \log n)$ 的排序算法有堆排序和归并排序。

(3) 空间复杂度为 $O(1)$ 的算法有简单选择排序、直接插入排序、冒泡排序、希尔排序和堆排序,空间复杂度为 $O(n)$ 的算法是归并排序,空间复杂度为 $O(\log n)$ 的算法是快速排序。

(4) 虽然直接插入排序和冒泡排序的速度比较慢,但是当初始序列整体或局部有序时,这两种排序算法会有较好的效率。当初始序列整体或局部有序时,快速排序算法的效率会下降。当排序序列较小且不要求稳定性时,直接选择排序效率较好;要求稳定性时,冒泡排序效率较好。

除了以上这几种排序算法以外,还有位图排序、桶排序、基数排序等。每种排序算法都有其最佳适用场合。例如,当待排序数据规模巨大,而对内存大小又没有限制时,位图排序则是最高效的排序算法。所以,在选择使用排序算法的时候,一定要结合实际情况进行分析。

13.6 二叉树

二叉树是一种非常常见并且实用的数据结构，它结合了有序数组与链表的优点。在二叉树中查找数据与在数组中查找数据一样快，在二叉树中添加、删除数据的速度也和链表中一样高效，所以有关二叉树的相关技术一直是程序员面试笔试中必考的知识点。

13.6.1 基础知识

二叉树 (Binary Tree) 也称为二分树、二元树、对分树等，它是 n ($n \geq 0$) 个有限元素的集合。该集合或者为空，或者由一个称为根 (root) 的元素及两个不相交的、被分别称为左子树和右子树的二叉树组成。当集合为空时，称该二叉树为空二叉树。

在二叉树中，一个元素也称为一个结点。二叉树的递归定义：二叉树或者是一棵空树，或者是一棵由一个根结点和两棵互不相交的分别称做根结点的左子树和右子树所组成的非空树，左子树和右子树又同样都是一棵二叉树。

以下是一些常见的二叉树的基本概念：

(1) 结点的度。结点所拥有的子树的个数称为该结点的度。

(2) 叶结点。度为 0 的结点称为叶结点，或者称为终端结点。

(3) 分枝结点。度不为 0 的结点称为分支结点，或者称为非终端结点。一棵树的结点除叶结点外，其余的都是分支结点。

(4) 左孩子、右孩子、双亲。树中一个结点的子树的根结点称为这个结点的孩子。这个结点称为它孩子结点的双亲。具有同一个双亲的孩子结点互称为兄弟。

(5) 路径、路径长度。如果一棵树的一串结点 n_1, n_2, \dots, n_k 有如下关系：结点 n_i 是 n_{i+1} 的父结点 ($1 \leq i < k$)，就把 n_1, n_2, \dots, n_k 称为一条由 $n_1 \sim n_k$ 的路径。这条路径的长度是 $k-1$ 。

(6) 祖先、子孙。在树中，如果有一条路径从结点 $M \sim$ 结点 N ，那么 M 就称为 N 的祖先，而 N 称为 M 的子孙。

(7) 结点的层数。规定树的根结点的层数为 1，其余结点的层数等于它的双亲结点的层数加 1。

(8) 树的深度。树中所有结点的最大层数称为树的深度。

(9) 树的度。树中各结点度的最大值称为该树的度，叶子结点的度为 0。

(10) 满二叉树。在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子结点都在同一层上，这样的一棵二叉树称为满二叉树。

(11) 完全二叉树。一棵深度为 k 的有 n 个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 i ($1 \leq i \leq n$) 的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。完全二叉树的特点是：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。需要注意的是，满二叉树肯定是完全二叉树，而完全二叉树不一定是满二叉树。

二叉树的基本性质如下：

性质 1：一棵非空二叉树的第 i 层上最多有 2^{i-1} 个结点 ($i \geq 1$)。

性质 2：一棵深度为 k 的二叉树中，最多具有 $2^k - 1$ 个结点，最少有 k 个结点。

性质 3：对于一棵非空的二叉树，度为 0 的结点（即叶子结点）总是比度为 2 的结点多一个，即如果叶子结点数为 n_0 ，度数为 2 的结点数为 n_2 ，则有 $n_0 = n_2 + 1$ 。

证明：用 n_0 表示度为 0（叶子结点）的结点总数，用 n_1 表示度为 1 的结点总数， n_2 表示度为 2 的结点总数， n 表示整个完全二叉树的结点总数，则 $n=n_0+n_1+n_2$ 。根据二叉树和树的性质，可知 $n=n_1+2\times n_2+1$ （所有结点的度数之和+1=结点总数），根据两个等式可知 $n_0+n_1+n_2=n_1+2\times n_2+1$ ，所以 $n_2=n_0-1$ ，即 $n_0=n_2+1$ 。所以 $n=n_0+n_1+n_2$ 。

性质 4：具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$ 。

证明：根据性质 2，深度为 k 的二叉树最多只有 2^k-1 个结点，且完全二叉树的定义是与同深度的满二叉树前面编号相同，即它的总结点数 n 位于 k 层和 $k-1$ 层满二叉树容量之间，即 $2^{k-1}-1 < n \leq 2^k-1$ 或 $2^{k-1} \leq n < 2^k$ ，三边同时取对数，于是有 $k-1 \leq \log_2 n < k$ ，因为 k 是整数，所以 $k = \lfloor \log_2 n \rfloor + 1$ 。

性质 5：对于具有 n 个结点的完全二叉树，如果按照从上至下和从左到右的顺序对二叉树中的所有结点从 1 开始顺序编号，则对于任意的序号为 i 的结点，有：（1）如果 $i>1$ ，则序号为 i 的结点的双亲结点的序号为 $i/2$ （其中“/”表示整除）；如果 $i=1$ ，则序号为 i 的结点是根结点，无双亲结点。（2）如果 $2i \leq n$ ，则序号为 i 的结点的左孩子结点的序号为 $2i$ ；如果 $2i > n$ ，则序号为 i 的结点无左孩子。（3）如果 $2i+1 \leq n$ ，则序号为 i 的结点的右孩子结点的序号为 $2i+1$ ；如果 $2i+1 > n$ ，则序号为 i 的结点无右孩子。

此外，若对二叉树的根结点从 0 开始编号，则相应的 i 号结点的双亲结点的编号为 $(i-1)/2$ ，左孩子的编号为 $2i+1$ ，右孩子的编号为 $2i+2$ 。

例题 1：一棵完全二叉树上有 1001 个结点，其中叶子结点的个数是多少？

分析：二叉树的公式： $n=n_0+n_1+n_2=n_0+n_1+(n_0-1)=2\times n_0+n_1-1$ 。而在完全二叉树中， n_1 只能取 0 或 1。若 $n_1=1$ ，则 $2\times n_0=1001$ ，可推出 n_0 为小数，不符合题意；若 $n_1=0$ ，则 $2\times n_0-1=1001$ ，则 $n_0=501$ 。所以答案为 501。

例题 2：如果根的层次为 1，具有 61 个结点的完全二叉树的高度为多少？

分析：根据二叉树的性质，具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$ ，因此含有 61 个结点的完全二叉树的高度为 $\lfloor \log_2 n \rfloor + 1$ ，即应该为 6 层。所以答案为 6。

例题 3：在具有 100 个结点的树中，其边的数目为多少？

分析：在一棵树中，除了根结点之外，每一个结点都有一条入边，因此总边数应该是 $100-1$ ，即 99 条。所以答案为 99。

13.6.2 如何递归实现二叉树的遍历

二叉树先序遍历的思想是从根结点开始，沿左子树一直走到没有左孩子的结点为止，依次访问所经过的结点，同时所经结点的地址进栈，当找到没有左孩子的结点时，从栈顶退出该结点的双亲的右孩子。此时，此结点的左子树已访问完毕，再用上述方法遍历该结点的右子树，如此重复到栈空为止。先序遍历的具体实现代码如下所示：

```
void PreOrder(BTree *tree)
{
    if(tree==NULL)
        return ;
    Operator(tree->data);
    if(tree->lchild!=NULL)
        PreOrder(tree->lchild);
    if(tree->rchild!=NULL)
        PreOrder(tree->rchild);
}
```

二叉树中序遍历的思想是从根结点开始，沿左子树一直走到没有左孩子的结点为止，并将所经结点的地址进栈，当找到没有左孩子的结点时，从栈顶退出该结点并访问它。此时，此结点的左子树已访问完毕，再用上述方法遍历该结点的右子树，如此重复到栈空为止。中序遍历的具体实现代码如下：

```
void MidOrder(BTree *tree)
{
    if(tree==NULL)
        return ;
    if(tree->lchild!=NULL)
        MidOrder(tree->lchild);
    Operator(tree->data);
    if(tree->rchild!=NULL)
        MidOrder(tree->rchild);
}
```

二叉树后序遍历的思想是从根结点开始，沿左子树一直走到没有左孩子的结点为止，并将所经结点的地址第一次进栈，当找到没有左孩子的结点时，此结点的左子树已访问完毕，从栈顶退出该结点，判断该结点是否为第一次进栈。如果是，再将所经结点的地址第二次进栈，并沿该结点的右子树一直走到没有右孩子的结点为止；如果不是，则访问该结点。此时，该结点的左右子树都已完全遍历，且令指针 $p = \text{NULL}$ ，如此重复直到栈空为止。后序遍历的具体实现代码如下：

```
void PostOrder(BTree *tree)
{
    if(tree==NULL)
        return ;
    if(tree->lchild!=NULL)
        PostOrder(tree->lchild);
    if(tree->rchild!=NULL)
        PostOrder(tree->rchild);
    Operator(tree->data);
}
```

13.6.3 已知先序遍历和中序遍历，如何求后序遍历

一般数据结构都有遍历操作，根据需求的不同，二叉树一般有以下几种遍历方式：先序遍历、中序遍历、后序遍历和层序遍历。

(1) 先序遍历：如果二叉树为空，遍历结束。否则，第一步，访问根结点；第二步，先序遍历根结点的左子树；第三步，先序遍历根结点的右子树。

(2) 中序遍历：如果二叉树为空，遍历结束。否则，第一步，中序遍历根结点的左子树；第二步，访问根结点；第三步，中序遍历根结点的右子树。

(3) 后序遍历：如果二叉树为空，遍历结束。否则，第一步，后序遍历根结点的左子树；第二步，后序遍历根结点的右子树；第三步，访问根结点。

(4) 层次遍历：从二叉树的第一层（根结点）开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。

图 13-15 中某二叉树结构图的先序遍历是 ABDHIEJCFG，中序遍历是 HDIBJEAF CG，后序遍历是 HIDJEBFGCA，层次遍历是 ABCDEFGHIJ。

例如，先序序列为 ABDECF，中序序列为 DBEAFC。首先先序遍历树的规则为根左右，可以看到先序遍历序列的第一个元素必为树的根结点，则 A 就为根结点。再看中序遍历为左

根右,再根据根结点 A,可知左子树包含元素为 DBE,右子树包含元素 FC。然后递归求解左子树(左子树的先序为 BDE,中序为 DBE),递归求解右子树(即右子树的先序为 CF,中序为 FC)。如此递归到没有左右子树为止。所以,树结构如图 13-16 所示。

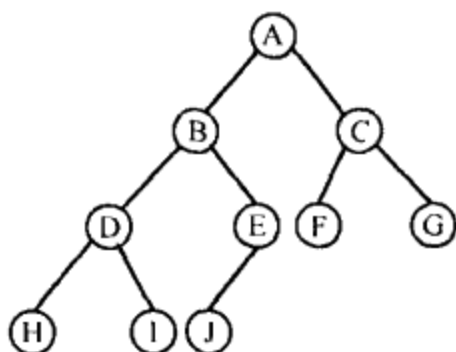


图 13-15 某二叉树结构图 (一)

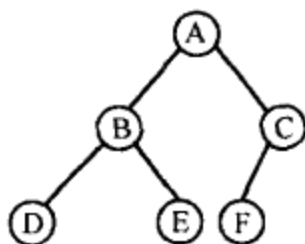


图 13-16 某二叉树结构图 (二)

通过上面的例子可以总结出用先序遍历和中序遍历来求解二叉树的过程,步骤如下:

(1) 确定树的根结点。树根是当前树中所有元素在先序遍历中最先出现的元素,即先序遍历的第一个结点就是二叉树的根。

(2) 求解树的子树。找到根在中序遍历的位置,位置左边是二叉树的左孩子,位置右边是二叉树的右孩子,若根结点左边或右边为空,则该方向子树为空;若根结点左边和右边都为空,则根结点已经为叶子结点。

(3) 对二叉树的左、右孩子分别进行步骤(1)、(2),直到求出二叉树结构为止。

具体实现代码如下:

```

int initTree(BTree root,char *front,char *middle,int num)
{
    int i=0;
    root->data=front[0];
    if(num==0) return 0;
    //找到根结点在 middle 的位置
    for(i=0;i<num;i++)
    {
        if(middle[i]==root->data) break;
    }
    //如果 root 存在左孩子
    if(i!=0)
    {
        root->lchild=new struct BTreeNode( );
        initTree(root->lchild,front+1,middle,i);
    }
    //如果 root 存在右孩子
    if(i!=num-1)
    {
        root->rchild=new struct BTreeNode( );
        initTree(root->rchild,front+i+1,middle+i+1,i);
    }
    return 1;
}
  
```

引申:假设一棵二叉树的后序遍历序列为 D G J H E B I F C A,中序遍历序列为 D B G E H J A C I F,则其先序遍历序列为多少?

本题中,可以首先确定 A 是根结点(在后序遍历的最后一个),再根据中序遍历的特点,可以知道 D B G E H J 为左子树, C I F 为右子树。再看右子树的后序遍历为 I F C,可以确定 C 为

右子树的根结点；再加上中序为 CIF，说明 C 无左子树，只有右子树。而左子树的后序遍历为 DGJHEB，因此 B 为左子树的根结点，再结合中序遍历，可以得知 B 的左子树只有 D，GEHJ 都是右子树。GEHJ 子树的后序遍历是 GJHE，说明 E 是根，HJ 为 E 的右子树，G 是 E 的左子树。最后可以确定 H 为 HJ 子树的根，J 为右子树。通过以上分析，就可以绘制出这棵树，如图 13-17 所示。

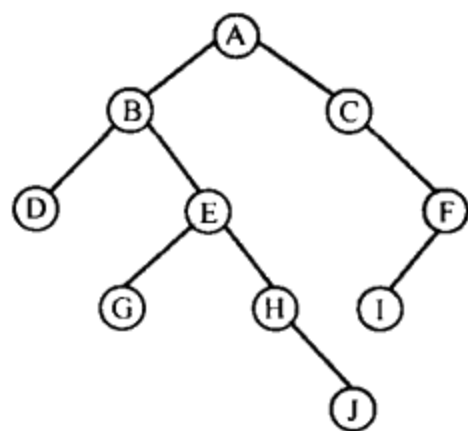


图 13-17 某二叉树结构图 (三)

所以，先序遍历为 ABDEGHJCFI。

通过上面的例子可以总结出用后序遍历和中序遍历来求解二叉树的过程：第一步确定树的根，树根是当前树中所有元素在后序遍历中最后出现的元素。第二步求解树的子树，找出根结点在中序遍历中的位置，根左边的所有元素就是左子树，根右边的所有元素就是右子树，如果根结点左边或右边为空，则该方向子树为空；若根结点左边和右边都为空，则根结点已经为叶子结点。第三步递归求解树，将左子树和右子树分别看成一棵二叉树。重复以上步骤，直到所有的结点完成定位。该过程与根据先序序列和中序序列求解树的过程类似，略有不同。

需要注意的是，如果知道先序与后序遍历序列，是无法构建二叉树的。例如，先序序列为 ABDECF，后序序列为 DEBFCA，此时只能确定根结点，而对于左右子树的组成不确定。

13.6.4 如何非递归实现二叉树的后序遍历

后序遍历可以用递归实现，程序中递归的调用就是保存函数的信息在栈中。一般情况下，能用递归解决的问题都可以用栈解决，只是递归更符合人们的思维方式，代码相对而言也更简单，但不能说明递归比栈的方式更快、更节省空间，因为在递归过程中都是操作系统来帮助用栈实现存储信息。下面用栈来实现二叉树的后序遍历。

栈的思想是“先进后出”，即首先把根结点入栈（这时栈中有一个元素），根结点出栈的时候再把它右左孩子入栈（这时栈中有两个元素，注意是“先进右后进左”，不是“先进左后进右”），再把栈顶出栈（也就是左孩子），再把栈顶元素的右左孩子入栈，此过程一直执行直到栈为空，出栈的元素按顺序排列就是这个二叉树的先序遍历。

用栈来解决二叉树的后序遍历是最后输出父亲结点，先序遍历是在结点出栈时入栈右左孩子。显然，对于后续遍历，不应该在父亲结点出栈时，才把右左孩子入栈，应该在入栈时就把右左孩子一并入栈。在父亲结点出栈时，应该判断右左孩子是否已经遍历过（是否执行过入栈），那么就应该有一个标记来判断孩子是否遍历过。下面借用二叉树的结构体来定义一个适用于这个算法的新结构体：

```
typedef struct stackTreeNode
{
    BTree treeNode;
    int flag;
} * pSTree;
```

结构体中，flag 为标志位，0 表示左右孩子没有遍历，2 表示左右孩子遍历完，具体实现代码如下：

```
int lastOrder(BTree root)
{
    stack<pSTree> stackTree;
    pSTree sTree = (pSTree)malloc(sizeof( struct stackTreeNode));
    sTree->treeNode=root;
    sTree->flag=0;
```

```

stackTree.push(sTree);
while(!stackTree.empty( ))
{
    pSTree tmpTree=stackTree.top( );
    if(tmpTree->flag==2)
    {
        cout<<tmpTree->treeNode->data<<" ";
        stackTree.pop( );
    }
    else
    {
        if(tmpTree->treeNode->rchild)
        {
            pSTree sTree = (pSTree)malloc(sizeof( struct stackTreeNode));
            sTree->treeNode=tmpTree->treeNode->rchild;
            sTree->flag=0;
            stackTree.push(sTree);
        }
        tmpTree->flag++;
        if(tmpTree->treeNode->lchild)
        {
            pSTree sTree = (pSTree)malloc(sizeof( struct stackTreeNode));
            sTree->treeNode=tmpTree->treeNode->lchild;
            sTree->flag=0;
            stackTree.push(sTree);
        }
        tmpTree->flag++;
    }
}
return 1;
}

```

引申：如何使用非递归方法实现二叉树的先序遍历与中序遍历？

将二叉树的先序遍历递归算法转化为非递归算法的方法如下：

- (1) 将二叉树的根结点作为当前结点。
- (2) 若当前结点非空，则先访问该结点，并将该结点进栈，再将其左孩子结点作为当前结点，重复步骤 (2)，直到当前结点为空为止。
- (3) 若栈非空，则栈顶结点出栈，并将当前结点的右孩子结点作为当前结点。
- (4) 重复步骤 (2)、(3)，直到栈为空且当前结点为空为止。

程序代码示例如下：

```

typedef struct
{
    Bitree Elem[100];
    int top;
}SqStack;

void PreOrderUnrec(Bitree t)
{
    SqStack s;
    StackInit(s);
    p=t;

    while (p!=null || !StackEmpty(s))
    {
        while (p!=null)

```

```

        {
            visite(p->data);
            push(s,p);
            p=p->lchild;
        }

        if (!StackEmpty(s))
        {
            p=pop(s);
            p=p->rchild;
        }
    }
}

```

将中序遍历递归算法转化为非递归算法的方法如下:

(1) 将二叉树的根结点作为当前结点。

(2) 若当前结点非空, 则该结点进栈并将其左孩子结点作为当前结点, 重复步骤 (2), 直到当前结点为空为止。

(3) 若栈非空, 则将栈顶结点出栈并作为当前结点, 接着访问当前结点, 再将当前结点的右孩子结点作为当前结点。

(4) 重复步骤 (2)、(3), 直到栈为空且当前为空为止。

程序代码示例如下:

```

typedef struct
{
    Bitree Elem[100];
    int top;
} SqStack;

void InOrderUnrec(Bitree t)
{
    SqStack s;
    StackInit(s);
    p=t;
    while (p!=null || !StackEmpty(s))
    {
        while (p!=null)
        {
            push(s,p);
            p=p->lchild;
        }

        if (!StackEmpty(s))
        {
            p=pop(s);
            visite(p->data);
            p=p->rchild;
        }
    }
}

```

13.6.5 如何使用非递归算法求二叉树的深度

计算二叉树的深度, 一般都是用后序遍历, 采用递归算法, 先计算出左子树的深度, 再算出右子树的深度, 最后取较大者加 1 即为二叉树的深度。算法示例如下:

```

typedef struct Node
{
    char data;
    struct Node *LChild;
    struct Node *RChild;
    struct Node *Parent;
}BNode,*BTree;

//后序遍历求二叉树的深度递归算法
int PostTreeDepth(BTree root)
{
    int leftheight,rightheight,max;
    if(root!=NULL)
    {
        leftheight=PostTreeDepth(root->LChild);
        rightheight=PostTreeDepth(root->RChild);
        max=leftheight>rightheight?leftheight:rightheight;
        return (max+1);
    }
    else
        return 0;
}

```

但如果直接将该算法改成非递归形式是非常繁琐和复杂的。考虑到二叉树深度与深度的关系，可以有下面两种非递归算法实现求解二叉树深度。

方法一：先将算法改成先序遍历再改写非递归形式。先序遍历算法：遍历一个结点前，先算出当前结点是在哪一层，层数的最大值就等于二叉树的深度。算法示例如下：

```

typedef struct Node
{
    char data;
    struct Node *LChild;
    struct Node *RChild;
    struct Node *Parent;
}BNode,*BTree;

int GetMax(int a,int b)
{
    return a>b?a:b;
}

int GetTreeTreeHeightPreorder(const BTree root)
{
    struct Info
    {
        const BTree TreeNode;
        int level;
    };
    deque<Info> dq;
    int level = -1;
    int TreeHeight = -1;
    while(1)
    {
        while(root)
        {
            ++level;
            if (root->RChild)
            {

```



```

        Info info = {root->RChild, level};
        dq.push_back(info);
    }
    root = root->LChild;
}
TreeHeight = GetMax (TreeHeight, level);
if (dq.empty( ))
    break;
const Info& info = dq.back( );
root = info.TreeNode;
level = info.level;
dq.pop_back( );
}
return TreeHeight;
}

```

方法二：修改上面提到的迭代算法。上例中，所用到辅助栈（或双端队列）的大小达到的最大值减去 1 就等于二叉树的深度。因而只需记录在往辅助栈放入元素后（或者在访问结点数据时），辅助栈的栈大小达到的最大值。算法示例如下：

```

typedef struct Node
{
    char data;
    struct Node *LChild;
    struct Node *RChild;
    struct Node *Parent;
}BNode,*BTree;

int GetMax(int a,int b)
{
    return a>b?a:b;
}

int GetTreeTreeHeightPostorder(const BTree root)
{
    deque<const BTree> dq;
    int TreeHeight = -1;
    while(1)
    {
        for ( ; root != NULL; root = root->LChild)
            dq.push_back(root);
        TreeHeight = GetMax(TreeHeight, (int)dq.size( ) - 1);
        while (1)
        {
            if (dq.empty( )) return TreeHeight;
            const BTree parrent = dq.back( );
            const BTree RChild = parrent->RChild;
            if (RChild && root != RChild)
            {
                root = RChild;
                break;
            }
            root = parrent;
            dq.pop_back( );
        }
    }
    return TreeHeight;
}

```

13.6.6 如何判断两棵二叉树是否相等

数据结构如下:

```
typedef struct _TreeNode
{
    char c;
    TreeNode *leftchild;
    TreeNode *rightchild;
}TreeNode;
```

函数接口为 int CompTree(TreeNode* tree1,TreeNode* tree2)。

注意: A、B 两棵树相等当且仅当 $rootA \rightarrow c == rootB \rightarrow c$, 而且 A 和 B 的左右子树相等或者左右互换相等。

可以采用递归的方式进行判断, 具体算法如下:

```
int compTree(TreeNode *tree1, TreeNode *tree2)
{
    if(!tree1 && !tree2)
        return 1;
    if((tree1 && !tree2) || (!tree1 && tree2))
        return 0;
    if(tree1 && tree2)
    {
        if(tree1->c==tree2->c)
        {
            if(compTree(tree1->leftChild, tree2->leftChild))
                return compTree(tree1->rightChild, tree2->rightChild);
            else if(compTree(tree1->rightChild, tree2->leftChild))
                return compTree(tree1->leftChild, tree2->rightChild);
        }
    }
    return 0;
}
```

对于上述算法, 在树的第 0 层, 有 1 个结点, 会进行 1 次函数调用; 在树的第 1 层, 有 2 个结点, 可能会进行 4 次函数调用; 在树的第 2 层, 有 4 个结点, 可能会进行 16 次函数调用.....在树的第 x 层, 有 2^x 个结点, 可能会进行 $(2^x)^2$ 次函数调用; 所以假设总结点数为 n, 则算法的复杂度为 $O(n^2)$ 。

13.6.7 如何判断二叉树是否是平衡二叉树

根据平衡二叉树的定义可知, 每个结点的左右子树的高度差小于等于 1, 只需在计算二叉树高度时, 同时判断左右子树的高度差即可。

所以可以采用递归的方式来判断, 算法如下:

```
int TreeHeight(const Node* root, bool& balanced)
{
    const int LHeight = root->left ? TreeHeight(root->left, balanced) + 1 : 0;
    if (!balanced)
        return 0;
    const int RHeight = root->right ? TreeHeight(root->right, balanced) + 1 : 0;
    if (!balanced)
        return 0;
    const int diff = LHeight - RHeight;
```

```

    if (diff < -1 || diff > 1)
        balanced = false;
    return (LHeight > RHeight ? LHeight : RHeight);
}

bool IsBalancedTree(const Node* root)
{
    bool balanced = true;
    if (root)
        TreeHeight(root, balanced);
    return balanced;
}

```

13.6.8 什么是霍夫曼编解码

在计算机中，数据是以 0、1 的形式进行存储和传递的，常见的字母、汉字、图片本质都是规则的二进制 01 串，这就涉及数据的编码。编码分为等长编码和非等长编码。ASCII 和 UNICODE 都是等长编码，等长编码存在一个局限，就是浪费空间。例如，给 4 个字母编码就需要两位，分别是 00、01、10 和 11，这样 0 和 1 这两个码字就没有意义。如果两位用来编码 6 个字母，就会出现这样一个问题：串 001011 无法解码出正确的意义，第一个 0 可能单独出现，也可能和后面的 0 成对出现。霍夫曼编码就是用来解决这种问题的。霍夫曼是一种非等长编码，其既不会引起歧义，又可以解码出正确的数据，并且出现概率大的数据编码短，概率小的数据编码长，极大地提高了编码效率。

霍夫曼编码用到一种叫做“前缀编码”的技术，即任意一个数据的编码都不是另一个数据编码的前缀。而最优二叉树，即霍夫曼树（带权路径长度最小的二叉树）就是一种实现霍夫曼编码的方式。霍夫曼编码的过程就是构造霍夫曼树的过程，构造霍夫曼树的相应算法如下：

（1）有一组需要编码且带有权值的字母，如 a(4)、b(8)、c(1)、d(2)、e(11)。括号内分别为各字母相对应的权值。

（2）选取字母中权值较小的两个 c(1)、d(2)组成一个新二叉树，其父亲结点的权值为这两个字母权值之和，记为 f(3)，然后将该结点加入到原字母序列中去（不包括已经选择的权值最小的两个字母），则剩下的字母为 a(4)、b(8)、e(11)、f(3)。此时得到的树如图 13-18 所示。

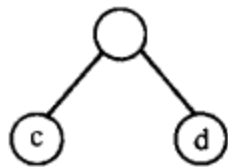


图 13-18 二叉树结构图（一）

（3）重复进行步骤（2），直到所有字母都加入到二叉树中为止，最后得到的二叉树如图 13-19 所示。

如果用 0 表示左分支，1 表示右分支，则得到的编码为 a(110)、b(10)、c(1110)、d(1111)、e(0)。

霍夫曼编码中所用数据结构如下：

```

typedef struct
{
    int *code; //结点编码
    char str; //结点的字符
}num; //结点的数据

```

```

typedef struct
{
    int weight,parent,lchild,rchild; //权值，父亲结点，左孩子和右孩子
    num ch; //结点数据
}

```

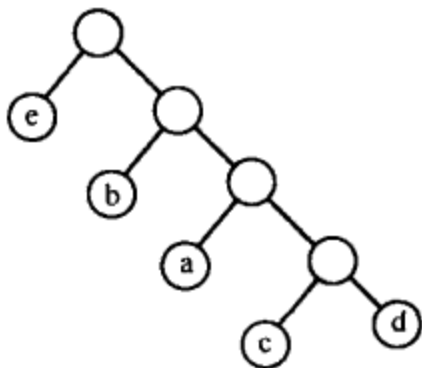


图 13-19 二叉树结构图（二）

}hnode,*tree; //结点

霍夫曼树的构造过程如下:

```
int CodingTree(tree &ht,int n)
{
    int m=2*n-1,*ful,i,s1,s2,f,c;
    ful=new int[n];
    for(i=n;i<m;i++)
    {
        select(ht,(i-1),s1,s2);//选择权值较小的两个结点
        ht[i].lchild=s1;
        ht[i].rchild=s2;
        ht[s1].parent=ht[s2].parent=i;
        ht[i].weight=ht[s1].weight+ht[s2].weight;
    }
    for(i=0;i<n;i++)
    {
        for(c=i,f=ht[i].parent,m=0;f!=-1;m++,c=f,f=ht[f].parent)
        {
            if(ht[f].lchild==c)
                ful[m]=0; //左孩子为 0
            else
                ful[m]=1; //右孩子为 1
        }
        m--;
        ht[i].ch.code=(int *)malloc(n*sizeof(int));
        for(int k=0;m>=0;k++,m--)
            ht[i].ch.code[k]=ful[m];
        ht[i].ch.code[k]='\n';
    }
    delete []ful;
    return 0;
}
```

霍夫曼树的解码过程与编码过程正好相反,从根结点出发,逐个读入编码内容:如果遇到0,则走左子树的根结点,否则走向右子树的根结点,一旦到达叶子结点,便译出代码所对应的字符。然后又重新从根结点开始继续译码,直到二进制编码结束。程序示例如下:

```
int DecodingTree(tree ht,int m,int *buff)
{
    int p=m-1;
    while(*buff!='\n')
    {
        if((*buff)==0)p=ht[p].lchild;
        else p=ht[p].rchild;
        buff++;
        if(ht[p].lchild==-1&&ht[p].rchild==-1)
        {
            cout<<ht[p].ch.str; // ht[p].ch.str 就是该编码所存储的数据
            p=m-1;
        }
    }
    return 0;
}
```

13.7 图

图论是计算机研究的一个重要分支,有关图论的内容可以写很多,但正是因为图论的这种

复杂性,在程序员面试笔试中,有关图论的问题并不多见,考查的也并不深奥。本节内容涉及一些经常出现的图论问题,并给予详细的解答。

13.7.1 什么是拓扑排序

从数学的角度来讲,拓扑排序就是由任意集合上的一个偏序关系得到一个该集合的全序关系的操作。如果将某一集合中的所有元素作为图的结点,将该集合上的偏序关系作为图的边,则任意一个偏序关系即可以表示一个有向图。

拓扑排序是有向图的一个重要操作。在给定的有向图 G 中,若顶点序列 v_1, v_2, \dots, v_n 满足下列条件:若在有向图 G 中从顶点 v_i 到顶点 v_j 有一条路径,则在序列中顶点 v_i 必在顶点 v_j 之前,便称这个序列为一个拓扑序列。求一个有向图拓扑序列的过程称为拓扑排序。

一个图的拓扑排序可以看成是图中所有顶点沿水平线排列而成的一个序列,使得所有的有向边均从左指向右。在很多应用中,有向无环图用于说明事件发生的先后次序。

常用的拓扑排序方法如下:

- (1) 从有向图中选择一个没有前驱(即入度为 0)的顶点并且输出它。
- (2) 从图中删去该顶点,并且删去从该顶点发出的所有边。
- (3) 重复上述步骤(1)和步骤(2),直到当前有向图中不存在没有前驱结点的顶点为止,或者当前有向图中的所有结点均已输出为止。

需要注意的是,一个有向无环图的拓扑排序序列不是唯一的。例如,对于图 13-20 而言,进行拓扑排序会得到两个序列: $\{v_1, v_2, v_5, v_4, v_3, v_7, v_6\}$ 或者 $\{v_1, v_2, v_5, v_4, v_7, v_3, v_6\}$ 。

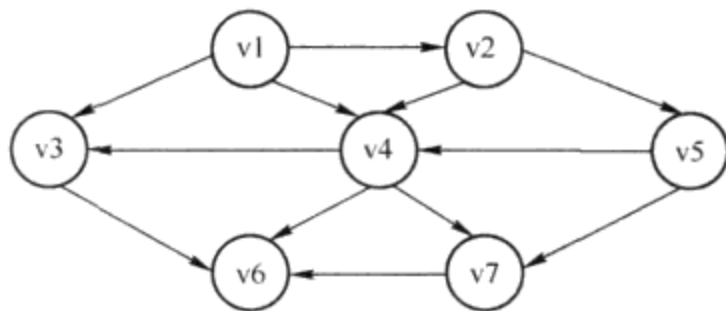


图 13-20 拓扑结构图

在现实的生活中,也有很多使用到拓扑排序的例子。例如,学校课程布置图,要先修完一些基础课,才可以继续修专业课,以计算机专业为例,在《程序设计基础》和《离散数学》课程学完之前就不能开始学习课程《数据结构》,这些先决条件定义了课程之间的领先(优先)关系。

具体实现代码如下:

```
status TopoLogicalSort (ALGraph G)
{
    //有向图 G 采用邻接表存储结构
    //若 G 无回路, 则输出 G 的顶点的一个拓扑序列并返回 ok, 否则返回 error
    FindIndegree(G, indegree);
    InitStack(s);
    for(i=0; i<G.vexnum; ++i)
        if(!indegree[i]) push(s, i);
    Count=0;
    While(!stackempty(s))
    {
        Pop(s, i); printf(G.vertices[i].data); ++count; //输出 i 号顶点并计数
        for(p=G.vertices[i].firstarc; p; p=p->next)
        {
            K=p->adjvex; //对 i 号顶点的每个邻接点的入度减 1
            if(!(--indegree[k])) push(s, k); //若入度减为 0, 则入栈
        }
    }
    if(count<G.vexnum)
```



```

        return error;    //该有向图有回路
    else
        return ok;
}

```

从拓扑排序的算法可知, 如果 AOV 网络有 n 个顶点, e 条边, 在拓扑排序的过程中, 搜索入度为零的顶点, 建立顶点栈所需要的时间是 $O(n)$ 。在正常的情况下, 有向图有 n 个顶点, 每个顶点进一次栈, 出一次栈, 共输出 n 次。顶点入度减 1 的运算共执行了 e 次。所以, 拓扑排序总的时间复杂度为 $O(n+e)$ 。

13.7.2 什么是 DFS? 什么是 BFS

最常见的图的遍历方式有深度优先遍历 (Depth First Search, DFS) 与广度优先遍历 (Breadth First Search, BFS) 两种。图的深度优先算法类似于树的先根遍历, 图的广度优先算法类似于树的层次遍历。

DFS 是从每一个顶点开始的深度优先遍历, 结果都是对该分支路径深入遍历到不能再深入为止, 且每个顶点只能被访问一次。具体实现过程为从图 G 中某个顶点 v 出发, 先访问该结点, 然后依次沿着未被访问过的 v 的邻接顶点进行深度优先遍历, 直到图 G 中和顶点 v 之间有相连路径的其他顶点都被访问过为止。此时, 如果图 G 中还有其他顶点未被访问过, 则从这些顶点中任选一个作为起始顶点, 重复上述过程, 直到图 G 中的所有顶点都被访问过为止。

例如, 图 13-21 是一个无向图, 假设从顶点 A 开始进行深度优先搜索, 则可能得到如下的一个访问过程: $A \rightarrow B \rightarrow E \rightarrow C \rightarrow F \rightarrow H \rightarrow G \rightarrow D$ 。其中, 当访问到顶点 E 时, 因为不存在未被访问过的 E 的邻接顶点, 所以沿着原路径回溯到 A (因为顶点 B 的所有邻接点都已经被访问过, 故直接回溯到顶点 A) 重新开始从顶点 C 开始搜索, 当沿着该条路径访问到顶点 D 时, 由于不存在未被访问过的 D 的邻接顶点, 故沿着原路径最终回溯到 A , 此时图中所有的顶点都已经被访问, 因此该图的深度优先搜索结束。

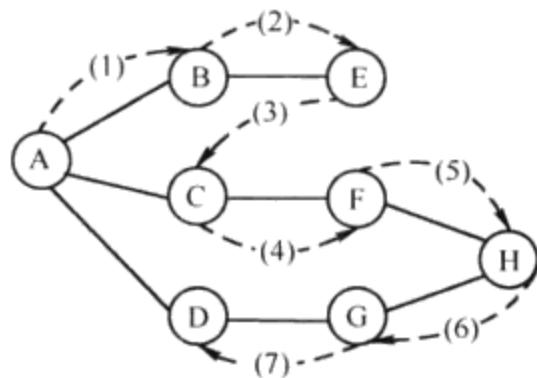


图 13-21 无向图

深度优先搜索算法的具体实现代码如下:

```

int visited[N];    //数组 visited[]表示图中顶点的访问情况, 1 表示已访问, 0 表示未访问
void DFS(Graph G,int v)
{
    visited[v]=1;
    Visit(v);    //函数 Visit(v)表示对顶点 v 的访问
    // 函数 FirstAdjVex(G,v)返回图 G 中 v 的第一个邻接顶点
    //函数 NextAdjVex(G,v,w)返回图 G 中 v 的 (相对于 w) 的下一个邻接顶点, 若 w 是 v 的最后一个邻
    接点, 则返回空
    for(w=FirstAdjVex(G,v),w>=0,w=NextAdjVex(G,v,w))
    {
        if(!visited[w])
            DFS(G,w)    //对 v 的未被访问过的邻接顶点进行深度优先搜索
    }
}

void DFSSearch(Graph G)    //对图 G 进行深度优先搜索
{
    for(v=0;v<G.vexnum;++v)
        visited[v]=0;
}

```

```

for(v=0;v<G.vexnum;++v)
    if(!visited[v])
        DFS(G,v);    //对未被访问过的顶点调用 DFS
}

```

BFS 也称为宽度优先算法, 属于一种盲目搜索方法, 是很重要的图算法的原型。其目的是逐层搜索图中的所有顶点, 且保证图中的所有顶点只被访问过一次。具体过程如下: 在给定图 $G=(V,E)$ 中, 从图 G 中的某个顶点 v 出发, 访问该顶点后, 依次访问所有未被访问过的 v 的邻接顶点, 然后再沿着这些顶点出发, 依次访问它们未被访问过的邻接顶点, 并且保证先被访问顶点的邻接顶点先于后被访问顶点的邻接顶点而被访问。所有与顶点 v 有相通路径的顶点都被访问结束后, 如果图 G 中还有其他顶点未被访问过, 则从这些顶点中任选一个顶点作为起始顶点, 重复上述过程, 直到图 G 中的所有顶点都被访问过为止。

例如, 图 13-22 是一个无向图, 假设从顶点 A 开始进行广度优先搜索, 则可能得到如下的一个访问过程: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H$ 。其中, 首先依次访问 A 的所有邻接顶点 B 、 C 、 D , 然后再依次访问 B 、 C 、 D 的所有未被访问过的邻接顶点, 最后再访问 F 的邻接顶点 H , 此时图中所有的顶点都已经被访问, 因此该图的广度优先搜索结束。

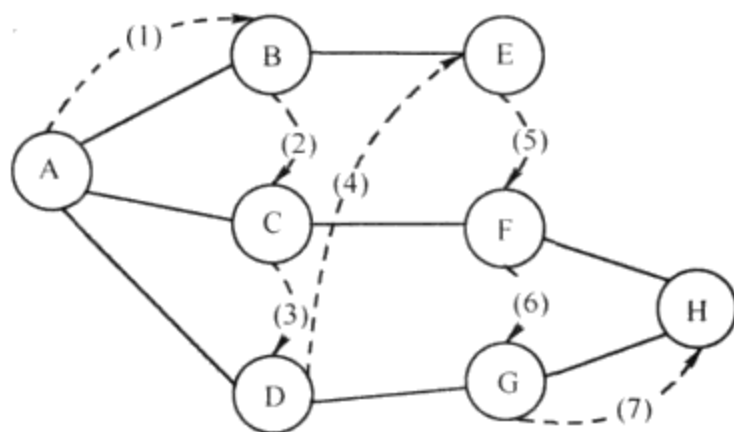


图 13-22 无向图

广度优先搜索算法的具体实现代码如下:

```

int visited[N]; //数组 visited[]表示图中顶点的访问情况, 1 表示已访问, 0 表示未访问
void BFSSearch(Graph G)
{
    for(v=0;v<G.vexnum;++v)
        visited[v]=0;
    InitQueue(Q)
    for(v=0;v<G.vexnum;++v)
        if(!visited[v])    //顶点 v 未被访问过
        {
            visited[v]=1;
            Visit(v);    //函数 Visit(v)表示对顶点 v 的访问
            EnQueue(Q,v);
            while(!QueueEmpty(Q))
            {
                DeQueue(Q,u);
                for(w=FirstAdjVex(G,u);w>=0;w=NextAdjVex(G,u,w))
                    if(!visited[w])    //w 是 u 的未被访问过的邻接顶点
                    {
                        visited[w]=1;
                        Visit(w);
                        EnQueue(Q,w);
                    }
            }
        }
}

```

13.7.3 如何求关键路径

图 13-23 所示是一个有 11 项活动的 AOE 网。其中有 9 个事件 v_1, v_2, \dots, v_9 , 每个事件表示

在它之前的活动已经完成,在它之后的活动可以开始。如 v_1 表示整个工程开始, v_9 表示整个工程结束, v_5 表示 a_4 和 a_5 已经完成, a_7 和 a_8 可以开始。与每个活动相联系的数是执行该活动所需的时间。

由于整个工程只有一个开始点和一个完成点,故在正常的情况(无环)下,网中只有一个入度为零的点(源点)和一个出度为零的点(汇点)。

由于在 AOE 网中有些活动可以并行地进行,所以完成工程的最短时间是从开始点到完成点的最长路径的长度。关键路径是指一个图中长度最长(路径上的各个活动所持续的时间之和)

的路径。用 $e(k)$ 表示活动(即弧) k 的最早开始时间,用 $l(k)$ 表示活动 k 的最晚开始时间,则把 $e(k)=l(k)$ 的活动叫做关键活动。关键路径上的所有活动都为关键活动。由于 AOE 网中的某些活动能够同时进行,故完成整个工程所必须花费的时间应该为始点到终点的最大路径长度。关键路径长度是整个工程所需的最短工期。

用 $ve(i)$ 表示事件(即顶点) i 的最早开始时间,用 $vl(i)$ 表示事件 i 的最晚开始时间。如果活动 k 由弧 $\langle m, n \rangle$ 表示,用 $dut(\langle m, n \rangle)$ 表示该活动的持续时间,则有:

$$e(k)=ve(m)$$

$$l(k)=vl(n)-dut(\langle m, n \rangle)$$

求解关键路径的具体算法如下(假设图中共有 n 个顶点):

(1) 从开始顶点 v_0 出发,假设 $ve(0)=0$,然后按照拓扑有序求出其他各顶点 i 的最早开始时间 $ve(i)$,如果得到的拓扑序列中顶点数目小于图中的顶点数,则表示图中存在回路,算法结束,否则继续执行。

(2) 从结束顶点 v_n 出发,假设 $vl(n-1)=ve(n-1)$,然后按拓扑有序求出其他各顶点 i 的最晚发生时间 $vl(i)$ 。

(3) 根据各顶点的最早开始时间 $ve(i)$ 和最晚开始时间 $vl(i)$ 依次求出每条弧的最早开始时间 $e(k)$ 和最晚开始时间 $l(k)$,如果有 $e(k)=l(k)$,则为关键活动。关键活动组成的路径则为关键路径。

具体实现代码如下:

```
void CriticalPath(Graph *G)
{
    int *etv,*ltv; //事件最早发生时间和最迟发生时间数组
    int top; //用于 Stack 的指针
    int *Stack; //用于存储拓扑序列的栈
    int ete,lte; //声明事件最早发生时间和最迟发生时间的变量
    TopoLogicalSort(G); //拓扑排序求事件的最早发生时间和拓扑序列 Stack
    ltv = (int *)malloc(sizeof(EdgeNode) * G->NumVertex);
    for(int i = 0; i < G->NumVertex; ++i) //初始化事件最晚发生时间
    {
        ltv[i] = etv[G->NumVertex - 1];
    }
    while(top != 0) //如果 Stack 栈不为空
    {
        int gettop = Stack[top--]; //出栈
        //处理下标为 gettop 的顶点所连接的顶点
```

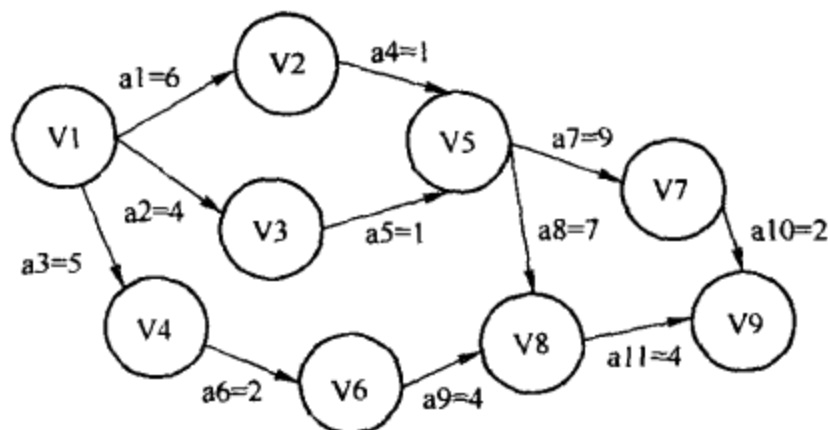


图 13-23 AOE 网

```

    for(EdgeNode *e = G->Vertex[gettop].FirstEdge;e=e->next)
    {
        int k = e->AdjVex;
        if(ltv[k] - e->weight < ltv[gettop])
            ltv[gettop] = ltv[k] - e->weight;
    }
}
for(int i = 0;i < G->NumVertex;++i)
{
    for(EdgeNode *e = G->Vertex[i].FirstEdge;e=e->next)
    {
        int k = e->AdjVex;
        ete = etv[i]; //事件最早发生时间
        lte = ltv[k] - e->weight; //事件最晚发生时间
        if(ete == lte) //相等即在关键路径上
        {
            printf("<V%d->V%d): %d\n",G->Vertex[i].data,G->Vertex[k].data,e->weight);
        }
    }
}
}
}

```

13.7.4 如何求最短路径

王老师家住在 A 地，一周内他需要对 B、C、D、E、F、G、H 七个地方的大学进行访问，A、B、C、D、E、F、G、H 这八个地点的位置及每两个地方之间的路径长度（两点之间边上的值表示这两点之间的路径长度），如图 13-24 所示。

由于从 A 地到其他各个地方的路径不止一条，为了减少出游的里程数，王老师需要事先计算出最优化路径。所以在访问前，王老师首先需要计算出从他所在的 A 地（源点）到其他各地方的最短路径（也就是路径长度之和最小），此时，Dijkstra（迪杰斯特拉）算法可以很好地解决这个问题。

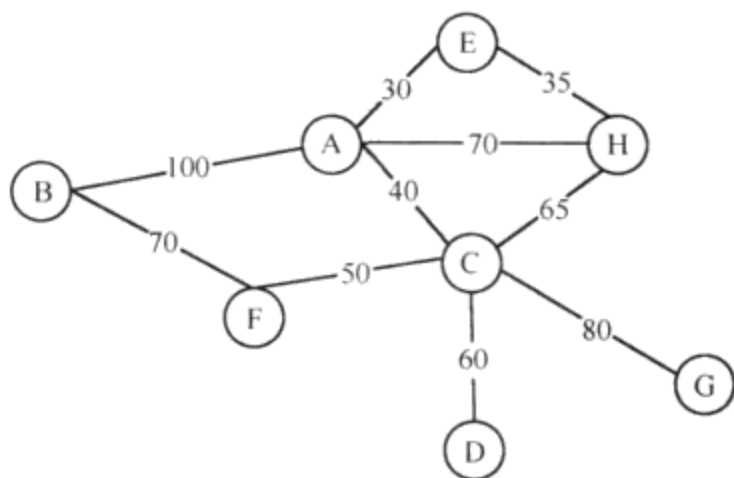


图 13-24 王老师家附近地形图

它的原理是对于源顶点 V_0 ，首先选择其直接相邻的顶点中长度最短的顶点 V_i ，那么根据已知可得，由 V_0 经过 V_i 到达与 V_i 直接相邻的顶点 V_j 的最短距离 $\text{dist}[j]$ 为 $\text{matrix}[V_0][j]$ 与 $\text{dist}[i] + \text{matrix}[i][j]$ 中的最小值，即 $\text{dist}[j] = \min\{\text{matrix}[V_0][j], \text{dist}[i] + \text{matrix}[i][j]\}$ 。

该算法的实现过程如下（设 S 是已求得最短路径的终点集合，V 是所有的结点集合）：

(1) $V - S$ = 未确定最短路径的顶点的集合，初始时 $S = \{A\}$ ，两个结点之间没有边时表示这两点之间的路径长度为无限大。

(2) 下一条路径的计算方式如下：首先，求出 A 到 V_i 中间只经 S 中顶点的最短路径，其中， $V_i \in V - S$ 。然后，将上述的最短路径与源点 A 到结点 V_i 的路径长度进行比较，长度最小者即为源点 A 到结点 V_i 的最短路径。最后将所求最短路径的终点（即 V_i ）加入 S 中。

(3) 重复步骤 (2)，直到求出所有终点的最短路径，即 S 中的结点数量与 V 中的结点数量相等。

Dijkstra 算法的时间复杂度为 $O(n^2)$ ，空间复杂度取决于存储方式，邻接矩阵为 $O(n^2)$ 。

具体算法如下所示:

```
typedef struct node
{
    int matrix[N][M];    //邻接矩阵
    int n;                //顶点数
    int e;                //边数
}MGraph;

void DijkstraPath(MGraph g,int *dist,int *path,int v0)
{
    int i,j,k;
    bool *visited=(bool *)malloc(sizeof(bool)*g.n);
    for(i=0;i<g.n;i++)    //初始化
    {
        if(g.matrix[v0][i]>0&&i!=v0)
        {
            dist[i]=g.matrix[v0][i];
            path[i]=v0;
        }
        else
        {
            dist[i]=INT_MAX;
            path[i]=-1;
        }
        visited[i]=false;
        path[v0]=v0;
        dist[v0]=0;
    }
    visited[v0]=true;
    for(i=1;i<g.n;i++)
    {
        int min=INT_MAX;
        int u;
        for(j=0;j<g.n;j++)
        {
            if(visited[j]==false&&dist[j]<min)
            {
                min=dist[j];
                u=j;
            }
        }
        visited[u]=true;
        for(k=0;k<g.n;k++)
        {
            if(visited[k]==false&&g.matrix[u][k]>0&&min+g.matrix[u][k]<dist[k])
            {
                dist[k]=min+g.matrix[u][k];
                path[k]=u;
            }
        }
    }
}
```

除了 Dijkstra 算法外, Bellman-Ford 算法也是一种常见的求解单源最短路径问题的算法。与 Dijkstra 算法不同的是, 在 Bellman-Ford 算法中, 边的权值可以为负数, 它的时效性比较好, 时间复杂度为 $O(VE)$ 。

计算机硬件的扩容确实可以极大地提高程序的处理速度，但考虑到其技术、成本等方面的因素，它并非一条放之四海而皆准的途径。而随着互联网技术的发展，云计算、物联网、移动通信技术的兴起，每时每刻，数以亿计的用户产生着数量巨大的信息，海量数据时代已经来临。由于通过对海量数据的挖掘能有效地揭示用户的行为模式，加深对用户需求的理解，提取用户的集体智慧，从而为研发人员决策提供依据，提升产品用户体验，进而占领市场，所以当前各大互联网公司研究都将重点放在了海量数据分析上，但是只寄希望于硬件扩容是很难满足海量数据分析需要的，如何利用现有条件进行海量信息处理已经成为各大互联网公司亟待解决的问题。所以，海量信息处理正日益成为当前程序员笔试面试中一个新的亮点。

14.1 问题分析

海量信息即大规模数据，随着互联网技术的发展，互联网上的信息越来越多，如何从海量信息中提取有用信息成为当前互联网技术发展必须面对的问题。

在海量数据中提取信息，不同于常规量级数据中提取信息，在海量信息中提取有用数据，会存在以下几个方面的问题：首先，数据量过大，数据中什么情况都可能存在，如果信息数量只有 20 条，人工可以逐条进行查找、比对，可是当数据规模扩展到上百条、数千条、数亿条，甚至更多时，仅仅只通过手工已经无法解决存在的问题，必须通过工具或者程序进行处理。其次，对海量数据信息处理，还需要有良好的软硬件配置，合理使用工具，合理分配系统资源。通常情况下，如果需要处理的数据量非常大，超过了 TB 级，小型机、大型工作站是要考虑的，普通的计算机如果有好的方法也可以考虑，如通过联机做成工作集群。最后，对海量数据信息处理时，要求很高的处理方法和技巧，如何进行数据挖掘算法的设计以及如何进行数据的存储访问等都是研究的难点。

本节的重点将放在如何运用好的方法和技巧来进行海量数据信息处理。

14.2 基本方法

针对海量数据的处理，可以使用的方法非常多，常见的方法有 Hash 法、Bit-map 法、Bloom filter 法、数据库优化法、倒排索引法、外排序法、Trie 树、堆、双层桶法以及 MapReduce 法。

1. Hash 法

Hash 一般被翻译为哈希，也被称为散列，它是一种映射关系，即给定一个数据元素，其关键字为 key，按一个确定的哈希函数 Hash 计算出 hash(key)，把 hash(key) 作为关键字 key 对应元素的存储地址（或称哈希地址），再进行数据元素的插入和检索操作。简而言之，哈希函数就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

哈希表是具有固定大小的数组，其中，表长（即数组的大小）应该为质数。哈希函数是用于关键字与存储地址之间的一种映射关系，但是不能保证每个元素的关键字与函数值是一一对应的，因为极有可能出现对应于不同的元素，却计算出了相同的函数值。冲突是指两个关键字映射到同一个存储地址的情况，即对不同的关键字可能得到同一散列地址，即 $\text{key1} \neq \text{key2}$ ，而 $f(\text{key1}) = f(\text{key2})$ 。

哈希函数一般应具备以下几个方面特点：

- 1) 运算应该尽可能简单。
- 2) 函数的值域必须在散列表的范围内。
- 3) 尽可能地减少冲突。

针对哈希函数的这些特点，在构建哈希表时，不仅要设定一个好的哈希函数，而且还要设定一种处理冲突的方法。常用的哈希函数的构建方法一般有以下几种：

(1) 直接寻址法。

取关键字或关键字的某个线性函数值为散列地址。即 $h(\text{key}) = \text{key}$ 或 $h(\text{key}) = a \cdot \text{key} + b$ ，其中 a 和 b 均为整型常数，这种散列函数叫做自身函数。例如，有一个人口数字统计表，人的年龄取值范围为 $1 \sim 100$ 岁，其中，年龄作为关键字，哈希函数取关键字自身，但这种方法效率比较低，时间复杂度为 $O(1)$ ，空间复杂度为 $O(n)$ ， n 为关键字的个数。

直接寻址法不会产生冲突，但由于它没有压缩映象，因此当关键字集合很大时，使用这种 Hash 函数是不可能实现地址编码的散列的。

(2) 取模法。

选择一个合适的正整数 p ，令 $\text{hash}(\text{Key}) = \text{Key} \bmod p$ 。 p 如果选择的是比较大的素数，则效果比较好，一般选取 p 为 TableSize，即哈希表的长度。

(3) 数字分析法。

设关键字是 d 位的以 r 为基的数（如以 10 为基的十进制数），且共有 n 个关键字。则关键字的每个位可能有 r 个不同的数符出现（即 $0, 1, 2, \dots, 9$ ），但这 r 个数符在各个位上出现的频率不一定相同，可能在某些位上分布比较均匀，即每个数符出现的次数接近于 n/r ，而在另一些位上分布不均匀。因此可选取其中分布比较均匀的那些位，重新组成新的数，用其作为哈希地址。

这种方法比较简单、直观，但是需要预先知道每个关键字的情况，这就限制了它的使用范围。

(4) 折叠法。

将关键字分成位数为 t 的几个部分（最后一部分的位数可能小于 t ），然后把各部分按位对齐进行相加，将所得的和舍弃进位，留下 t 位作为哈希地址。当关键字位数很多，而且关键字中每位上数字分布比较均匀时，采用折叠法比较合适。

(5) 平方取中法。

这是一种较常用的方法，将关键字进行平方运算，然后从结果的中间取出若干位（位数与散列地址的位数相同），将其作为散列地址，具体取几位由哈希表的表长决定。

(6) 除留余数法。

除留余数法是一种比较常用的哈希函数，它的主要原理是取关键字除以某个数 p （ p 不大于哈希表的长度 TableSize）的余数作为哈希地址，即 $\text{Hash}(\text{key}) = \text{key} \% p$ 。

使用除留余数法时，选取合适的 p 值很重要，一般要求 $p \leq \text{TableSize}$ ，且接近 TableSize。

或等于 TableSize, p 一般选取质数, 也可以是不包含小于 20 质因子的合数。

(7) 随机数法。

选择一个随机函数, 然后用关键字 key 的随机函数值作为哈希地址, 即 $Hash(key) = random(key)$;

其中, $random()$ 为随机函数。当关键字的长度不相等时采用这种方法比较合适。

在构造哈希表的过程中, 不管使用什么样的哈希函数, 冲突都不可能完全避免的, 所以冲突解决是构造哈希表的一个必不可少的过程。解决冲突的主要途径是当一个关键字映射到哈希表中的某一个地址且该地址上已有关键字时, 再为该关键字寻找新的存储地址。常用的冲突解决办法有以下几种:

(1) 开放地址法。

开放地址法的基本思想是当发生地址冲突时, 则在哈希表中再按照某种方法继续探测其他的存储地址, 直到找到空闲的地址为止。该过程可描述为

$$H_i(key) = (H(key) + d_i) \bmod m \quad (i=1, 2, \dots, k \ (k \leq m-1))$$

其中, $H(key)$ 为关键字 key 的直接哈希地址, m 为哈希表的长度, d_i 为每次再探测时的地址增量。

采用这种方法时, 首先计算出关键字的直接哈希地址, 即 $H(key)$, 如果该直接哈希地址上已经有其他的关键字, 则继续查看地址为 $[H(key) + d_i]$ 的存储地址, 判断其是否为空。如此反复直至找到空闲的存储地址为止, 然后将关键字 key 存放到该地址。

增量 d_i 可以有不同的取法, 常用的有以下 3 种:

- 1) $d_i = 1, 2, 3, \dots, m-1$, 称为线性探测再散列。
- 2) $d_i = 12, -12, 22, -22, \dots, -k2 \ (k \leq m/2)$, 称为二次探测再散列。
- 3) d_i = 伪随机序列, 称为伪随机再散列。

注意: 对于利用开放地址法处理冲突所产生的哈希表中, 删除一个元素时不能直接删除, 因为这样将会影响其他具有相同哈希地址的元素的查找地址, 所以通常采用设定一个特殊的标志的方法表示该元素已经被删除。

(2) 链地址法。

链地址法解决冲突的主要思想是: 如果哈希表空间为 $[0, m-1]$, 则设置一个由 m 个指针组成的一维数组 $CH[m]$, 然后在寻找关键字哈希地址的过程中, 所有哈希地址为 i 的数据元素都插入到头指针为 $CH[i]$ 的链表中。这种方法比较适合于冲突比较严重的情况下使用。

例如, 设有 8 个元素 $\{a, b, c, d, e, f, g, h\}$, 采用某种哈希函数得到的地址分别为 $\{0, 2, 4, 1, 0, 8, 7, 2\}$, 当哈希表长度为 10 时, 采用链地址法解决冲突的哈希表如图 14-1 所示。

(3) 再散列法。

当发生冲突时, 使用第二个、第三个哈希函数计算地址, 直到无冲突时。但这种方法的缺点是计算时间会大幅增加。

(4) 建立一个公共溢出区。

假设哈希函数的值域为 $[0, m-1]$, 则设向量 $HashTable[0, \dots, m-1]$ 为基本表, 另外设立存储空

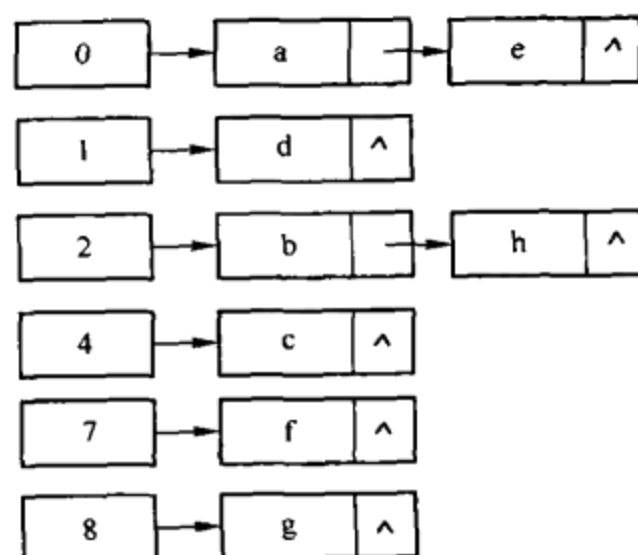


图 14-1 链地址法解决冲突法

间向量 $OverTable[0, \dots, v]$ 用以存储发生冲突的记录。

Hash 主要是用来进行“快速存取”，在 $O(1)$ 时间复杂度里就可以查找到目标元素，或者判断其是否存在。Hash 数据结构里的数据对外是杂乱无序的，无法得知其具体存储位置，也不知道各个存储元素位置之间的相互关系，但是却可以在常数时间里判断元素位置及存在与否。在海量数据处理中，使用 Hash 方法一般可以快速存取、统计某些数据，将大量数据进行分类。例如，提取某日访问网站次数最多的 IP 地址等。

2. Bit-map 法

Bit-map（位图）法的基本原理是使用位数组来表示某些元素是否存在，如 8 位电话号码中查重复号码，它适用于海量数据的快速查找、判重、删除等。

具体而言，位图排序以一个 N 位长的串，每位上以“1”或“0”表示需要排序的集合（后简称“集合”）中的数。例如，集合为 $\{2, 7, 4, 9, 1, 10\}$ ，则生成一个 10 位的串，将第 2、7、4、9、1、10 位置为“1”，其余位置为“0”，这样当把串中所有位都置完后，排序也自动完成了（因为字符串的下标是有序的）：1101001011。

再例如要排序 0~15 内的以下元素序列 $\{5, 8, 1, 12, 6, 2\}$ ，那么首先开辟两个字节的空間，也就是 16 位，分别对应 0~15 这 16 个数。首先，将这 16 位置为 0。遍历序列，在出现的数字的对应位置上置 1，也就是将每个元素对应到了位图的相应位置。再遍历这 16 位，就完成了对元素的排序，过程如图 14-2 所示。

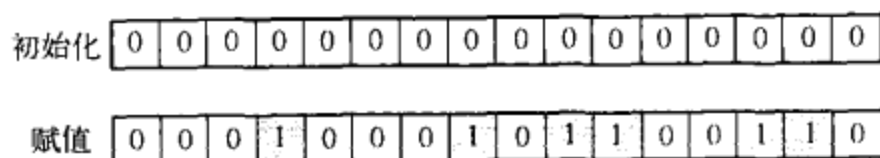


图 14-2 位图法初始化与赋值

位图排序的时间复杂度是 $O(n)$ 的，比一般的排序都快，但它是用空间换时间（需要一个 N 位的串）的，而且有一些限制，即数据状态不是很多。例如，排序前集合大小最好已知，而且集合中元素的最大重复次数必须已知，最好是稠集数据（不然空间浪费很大）。

在程序设计中，经常会遇到判断集合中是否存在重复的问题，数据量比较小时，对时间复杂度可能要求并不高，但当集合中数据量比较大时，则希望能够少进行几次扫描，此时如果还采用双重循环法，显然效率就太低下了，不可取。而位图法则比较适合于这种情况，位图法首先扫描一遍集合，找出集合中的最大元素，然后按照集合中最大元素 max 创建一个长度为 $max+1$ 的新数组，接着再次扫描原数组，每遇到一个元素，就将新数组中下标为元素值的位上置 1。例如，如果遇到元素 5，就将新数组的第 6 个元素置为 1，如此下去，当下次再遇到元素 5 想置位时，发现新数组的第 6 个元素已经被置为 1 了，则说明这次的数据肯定和以前的数据存在着重复。该算法的运算次数最坏的情况为 $2N$ ，但如果能够事先知道集合的最大元素的值，那么效率还可以提高一倍。

位图法的作用巨大，除了判断数据是否重复以外，也经常使用位图法来判断集合中某个数据是否存在。

3. Bloom filter 法

日常生活中很多地方都会遇到类似这样的问题：在设计计算机软件系统时，在程序中经常需要判断一个元素是否在一个集合中；在字处理软件中，需要检查一个英语单词是否拼写正确；在 FBI，一个嫌疑人的名字是否已经在嫌疑名单上。

针对这些问题，最直接的解决方法就是将集合中全部的元素都存储在计算机中，每当遇到

一个新元素时，就将它和集合中的元素直接进行比较即可。这种做法虽然能够准确无误地完成任任务，但存在一个问题，就是比较次数太多，效率比较低。当数据量不大时，这种效率低的问题并不显著，但是当数据量巨大时，如在海量数据信息处理中，存储效率低的问题就显现出来了。

Bloom filter 正是解决这一问题的有效方法，它是一种空间效率和时间效率很高的随机数据结构，它用来检测一个元素是否属于一个集合。但它同样带来一个问题：牺牲了正确率，Bloom filter 以牺牲正确率为前提，来换取空间效率与时间效率的提高。当它判断某元素不属于这个集合时，该元素一定不属于这个集合；当它判断某元素属于这个集合时，该元素不一定属于这个集合。具体而言，查询结果有两种可能，即“不属于这个集合（绝对正确）”和“属于这个集合（可能错误）”。所以，Bloom filter 适合应用在对于低错误率可以容忍的场合。

它的基本原理是位数组与 Hash 函数的联合使用。具体而言，首先，Bloom filter 是一个包含了 m 位的位数组，数组的每一位都初始化为 0，然后定义 k 个不同的 Hash 函数，每个函数都可以将集合中的元素映射到位数组的某一位。当向集合中插入一个元素时，根据 k 个 Hash 函数可以得到位数组中的 k 个位，将这些位设置为 1。如果查询某个元素是否属于集合，那么根据 k 个 Hash 函数可以得到位数组中的 k 个位，查看这 k 个位中的值，如果有的位不为 1，那么该元素肯定不在此集合中；如果这 k 个位全部为 1，那么该元素可能在此集合中（在插入其他元素时，可能会将这些位置为 1，这样就产生了错误）。

下面通过一个实例具体了解 Bloom filter，如图 14-3 所示。

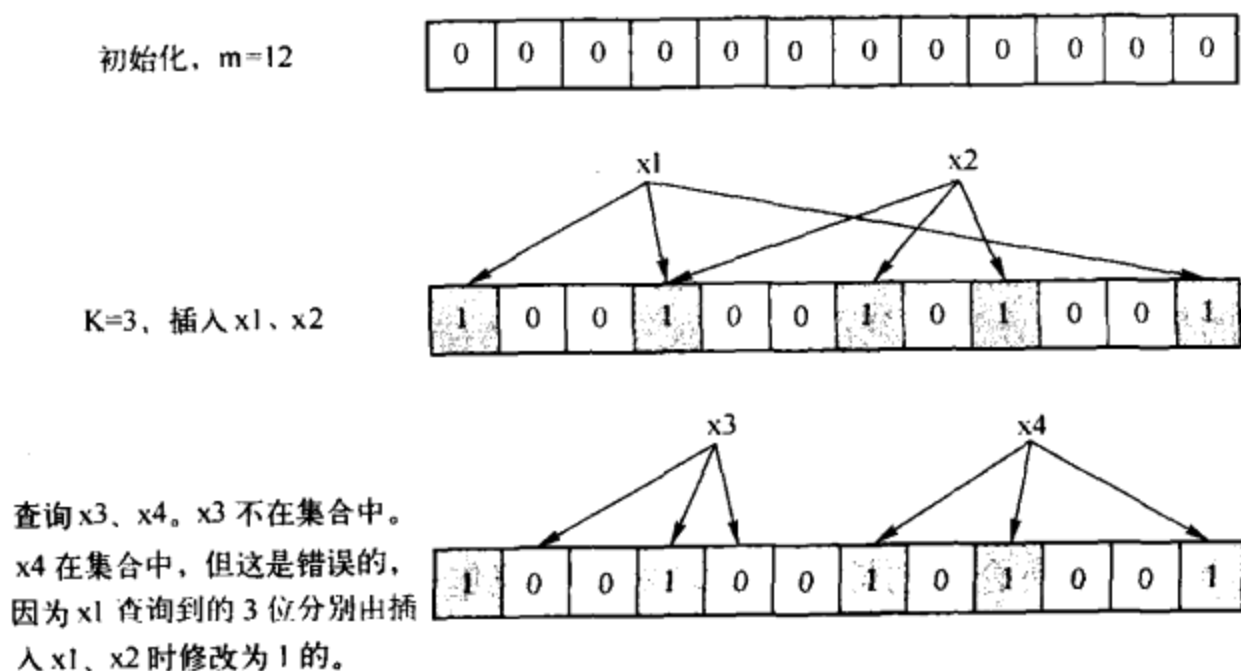


图 14-3 Bloom filter 实例解析

所以，使用 Bloom filter 的难点是如何根据输入元素个数 n ，来确定位数组 m 的大小以及 Hash 函数。当 Hash 函数个数 $k=(\ln 2) \times (m/n)$ 时错误率最小，在错误率不大于 E 的情况下， m 至少要等于 $n \times \lg(1/E)$ 才能表示任意 n 个元素的集合。但 m 还应该更大些，因为还要保证位数组里至少一半为 0，则 m 应该 $\geq n \lg(1/E) \times \lg e$ ，大概就是 $n \lg(1/E)$ 的 1.44 倍 (\lg 表示以 2 为底的对数)。

例如，假设 E 为 0.01，即错误率为 0.01，则此时 m 应该大约为 n 的 13 倍。这样 k 大约是 8 个（注意， m 与 n 的单位不同， m 的单位是 bit，而 n 则是以元素个数为单位）。通常单个元素的长度都是有很多 bit 的，所以使用 bloom filter 内存上通常都是节省的。

Bloom filter 的优点是具有很好的空间效率和时间效率。它的插入和查询时间都是常数，

另外它不保存元素本身，具有良好的安全性。然而，这些优点都是以牺牲正确率为代价的。当插入的元素越多，错判“元素属于这个集合”的概率就越大。另外，Bloom filter 只能插入元素，却不能删除元素，因为多个元素的哈希结果可能共用了 Bloom filter 结构中的同一个位，如果删除元素，就可能会影响多个元素的检测。所以，Bloom filter 可以用来实现数据字典、进行数据的判重或者集合求交集。

CBF 与 SBF 是 BF 的扩展，Counting Bloom Filter (CBF) 将位数组中的每一位扩展为一个 counter，从而支持了元素的删除操作。Spectral Bloom Filter (SBF) 将其与集合元素的出现次数关联，SBF 采用 counter 中的最小值来近似表示元素的出现频率。

4. 数据库优化法

互联网上的数据一般都被存储在数据库中，很多情况下，人们并非对这些海量数据本身感兴趣，而是需要从这些海量数据中提起出对自己有用的信息。例如，从数据中获取访问最多的页面信息等，这就涉及数据的查询技术等相关内容。

数据库管理软件选择是否合理、表结构设计是否规范、索引创建是否恰当都是影响数据库性能的重要因素。所以，对数据库进行优化，是实现海量数据高效处理的有效方法之一。常见的数据库优化方法有以下几种：

(1) 优秀的数据库管理工具。

选择一款优秀的数据库管理工具非常重要。现在的数据库工具厂家比较多，对海量数据的处理对所使用的数据库工具要求比较高，一般使用 Oracle、DB2、MySQL 等。

(2) 数据分区。

进行海量数据的查询优化，一种重要方式就是如何有效地存储并降低需要处理的数据规模，所以可以对海量数据进行分区操作提高效率。例如，针对按年份存取的数据，可以按年进行分区，不同的数据库有不同的分区方式，不过处理机制却大体相同。例如，SQL Server 的数据库分区是将不同的数据存于不同的文件组下，而不同的文件组存于不同的磁盘分区下，这样将数据分散开，减小磁盘 I/O，减小了系统负荷，而且还可以将日志、索引等放于不同的分区下。

(3) 索引。

索引一般可以加速数据的检索速度，加速表与表之间的链接，提高性能，所以在对海量数据进行处理时，考虑到信息量比较大，应该对表建立索引，包括在主键上建立聚簇索引，将聚合索引建立在日期列上等。

索引优点很多，但是对于索引的建立，还需要考虑到实际情况，而不是对每一个列建立一个索引。例如，针对大表的分组、排序等字段，都要建立相应的索引，同时还应该考虑建立复合索引。增加索引同时也有很多不利的方面：首先，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加；其次，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间。如果要建立聚簇索引，那么需要的空间就会更大。最后，当对表中的数据进行增加、删除和修改的时候，索引也要动态地维护，这样就降低了数据的维护速度。

所以索引要用到好的时机，索引的填充因子和聚集、非聚集索引都要考虑。

(4) 缓存机制。

当数据量增加时，一般的处理工具都要考虑到缓存问题。缓存大小设置的好差也关系到数据处理的成败。例如，在处理 2 亿条数据聚合操作时，缓存设置为 100000 条/Buffer 可行。

(5) 加大虚存。

由于系统资源有限，而需要处理的数据量非常大，所以当内存不足时，可以通过增加虚拟内存来解决。

(6) 分批处理。

由于需要处理的信息量巨大，可以对海量数据进行分批处理（类似于云计算中的 MapReduce 思想），然后再对处理后的数据进行合并操作，分而治之，有利于小数据量的处理，不至于面对大数据量带来的问题。

(7) 使用临时表和中间表。

数据量增加时，处理中要考虑提前汇总。这样做的目的是化整为零，大表变小表，分块处理完成后，再利用一定的规则进行合并，处理过程中的临时表的使用和中间结果的保存都非常重要。如果对于超海量的数据，大表处理不了，只能拆分为多个小表。如果处理过程中需要多步汇总操作，可按汇总步骤一步步来。

(8) 优化查询语句。

查询语句的性能对查询效率的影响是非常大的。编写高效优良的 SQL 脚本和存储过程是数据库工作人员的职责，也是检验数据库工作人员水平的一个标准。

(9) 使用视图。

视图中的数据来源于基本表，对海量数据的处理，可以将数据按一定的规则分散到各个基本表中，查询或处理过程中可以基于视图进行。

(10) 使用存储过程。

在存储过程中尽量使用 SQL 自带的返回参数，而非自定义的返回参数，减少不必要的参数，避免数据冗余。

(11) 用排序来取代非顺序存取。

磁盘存取臂的来回移动使得非顺序磁盘存取变成了最慢的操作，但是在 SQL 语句中这个现象被隐藏了，这样就使得查询中进行了大量的非顺序页查询，降低了查询速度。

(12) 使用采样数据进行数据挖掘。

基于海量数据的数据挖掘正在逐步兴起，面对着超海量的数据，一般的挖掘软件或算法往往采用数据抽样的方式进行处理，这样的误差不会很高，大大提高了处理效率和处理的成功率。一般采样时要注意数据的完整性，防止过大的偏差。

5. 倒排索引法

倒排索引是目前搜索引擎公司对搜索引擎最常用的存储方式，也是搜索引擎的核心内容。在搜索引擎实际的引用之中，有时需要按照关键字的某些值查找记录，所以是按照关键字建立索引，这个索引就被称为倒排索引。

倒排索引也常被称为反向索引、置入档案或反向档案，它本质上是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构，有两种不同的反向索引形式：第一种形式是一条记录的水平反向索引（或者反向档案索引）包含每个引用单词的文档的列表；第二种形式是一个单词的水平反向索引（或者完全反向索引）又包含每个单词在一个文档中的位置。第二种形式提供了更多的兼容性（如短语搜索），但是需要更多的时间和空间来创建。

一般情况下可以采用矩阵的方式来存储，但会浪费大量的空间。例如，对于如下的内容，

D1: The GDP increased.
D2: The text is this.
D3: My name is.

如果采用矩阵的方式存储, 见表 14-1。其中, 行表示关键词, 列表示所有的文件。

表 14-1 矩阵方式存储表示

| | D1 | D2 | D3 |
|-----------|----|----|----|
| The | 1 | 1 | 0 |
| GDP | 1 | 0 | 0 |
| increased | 1 | 0 | 0 |
| text | 0 | 1 | 0 |
| is | 0 | 1 | 1 |
| name | 0 | 0 | 1 |

而根据表 14-1, 就能得到下面的倒排索引:

```
The: {D1, D2};
GDP: {D1};
increased: {D1};
Text: {D2};
is: {D2, D3};
Name: {D3}.
```

通过比较发现, 采用倒排索引比采用矩阵的方式节省很多的空间。

正向索引开发出来用来存储每个文档的单词的列表。正向索引的查询往往满足每个文档有序频繁的全文查询和每个单词在校验文档中的验证查询。在正向索引中, 文档占据了中心的位置, 每个文档指向了一个它所包含的索引项的序列。也就是说, 文档指向了它包含的那些单词, 而反向索引则是单词指向了包含它的文档, 很容易看到这个反向的关系。而与正向索引相比, 倒排索引的优点是在处理复杂的多关键字查询时, 可在倒排表中先完成查询的并、交等逻辑运算, 得到结果后再对记录进行存取, 这样不必对每个记录随机存取, 把对记录的查询转换为地址集合的运算, 从而提高查找速度。所以, 倒排索引一般被应用于文档检索系统, 查询哪些文件包含了某个单词, 比如常见的学术论文的关键字搜索。

6. 外排序法

当待排序的对象数目特别多时, 在内存中不能一次处理, 必须把它们以文件的形式存放于外存, 排序时再把它们一部分一部分调入内存进行处理, 该种方式就是外排序。

外排序是相对内排序而言的, 它是大文件的排序, 待排序的记录存储在外存储器上, 待排序的文件无法一次装入内存, 需要在内存和外部存储器之间进行多次数据交换, 以达到排序整个文件的目的。一般采用归并排序等方式实现外排序, 主要分成两个步骤: 第一步, 生成若干初始归并段 (顺串), 也被称为文件预处理, 把含有 n 个记录的文件, 按内存大小划分为若干长度为 L 的子文件, 然后分别将子文件调入内存, 采用有效的内排序方法排序后送回外存; 第二步进行多路归并, 即对这些初始归并段进行多遍归并, 使得有序的归并段逐渐扩大, 最后在外存上形成整个文件的单一归并段, 此时就完成了文件的外排序。

外排序的适用范围是大数据的排序以及去重复。但外排序也存在着很大的缺陷, 就是它会消耗大量的 IO, 效率不会很高。

7. Trie 树

Trie 这个单词来自于 “retrieve”, Trie 树又称字典树或键树。它是一种用于快速字符串检索的多叉树结构, 其原理是利用字符串的公共前缀来降低时空开销, 即以空间换时间, 从而达到提高程序效率的目的。Trie 树的典型应用是用于统计和排序大量的字符串 (但不仅限于字符

串), 所以经常被搜索引擎系统用于文本词频统计。它的优点是: 最大限度地减少无谓的字符串比较, 查询效率比哈希表高。

Trie 树一般具有以下 3 个基本特性:

(1) 根结点不包含字符, 除根结点外每一个结点都只包含一个字符。

(2) 从根结点到某一结点, 路径上经过的字符连接起来, 为该结点对应的字符串。

(3) 每个结点的所有子结点包含的字符都不相同。

Trie 树可以利用字符串的公共前缀来节约存储空间。

如图 14-4 所示, 该 Trie 树用 10 个结点保存了 5 个字符串 amy、ann、em、rob、rg。

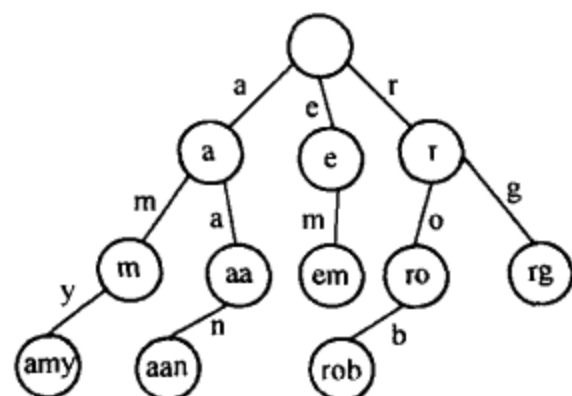


图 14-4 公共前缀图

在该 Trie 树中, 字符串 “amy” 和 “ann” 有公共前缀 “a”。当然, 如果系统中存在大量字符串且这些字符串基本没有公共前缀, 则相应的 Trie 树将非常消耗内存, 这也是 Trie 树的一个缺点。

例如, 给一个单词 a, 如果通过交换单词中字母的顺序可以得到另外的单词 b, 那么称 b 是 a 的兄弟单词。例如, 单词 army 和 mary 互为兄弟单词。现在要给出一种解决方案, 对于用户输入的单词, 根据给定的字典找出输入单词有哪些兄弟单词。

一般情况下, Trie 树的结构都是采用 26 叉树进行组织的, 每个结点对应一个字母, 查找的时候, 就是一个字母一个字母地进行匹配, 算法的时间复杂度就是单词的长度 n, 效率很高。本例子可以定义一个 Trie 树作为数据结构来查询, 此时就转化为在一棵 Trie 树中查找兄弟单词, 只要在 Trie 树中的前缀中再存储一个 vector 结构的容器, 就可以大大降低时间复杂度。

具体求解兄弟单词的程序代码如下:

```
#include <iostream>
#include <vector>
#include <string>
#include <stdlib.h>
using namespace std;

struct TrieNode
{
    vector<string> bwords;
    TrieNode *next[26];
    TrieNode()
    {
        for (int i = 0; i < 26; i++)
        {
            next[i] = NULL;
        }
    }
};

int CmpChar(const void *arg1, const void *arg2)
{
    return (*(char *)arg1) - (*(char *)arg2);
}

void InsertNode(TrieNode **root, string wd)
{

```

```

    if (wd.size() == 0)
    {
        return;
    }
    string t;
    //VC6 的 string 拷贝构造函数采用引用计数, 如果两个串一样的就不复制。为了使两个字符串指向的位置不一样
    t += wd;
    char *swd = const_cast<char*>(t.c_str());
    qsort(swd, wd.size(), sizeof(char), CmpChar);
    if (*root == NULL)
    {
        *root = new TrieNode();
    }

    int i = 0;

    TrieNode * next = *root;
    while (i < wd.size())
    {
        if (next->next[swd[i]-'a'] == NULL)
        {
            TrieNode * nn = new TrieNode();
            next->next[swd[i]-'a'] = nn;
        }

        next = next->next[swd[i]-'a'];
        i++;
    }
    next->bwords.push_back(wd);
}

bool SearchNode(TrieNode *root, string wd)
{
    char *swd = const_cast<char*>(wd.c_str());
    qsort(swd, wd.size(), sizeof(char), CmpChar);
    int i = 0;
    while (i < wd.size())
    {
        if (root->next[swd[i]-'a'] != NULL)
        {
            root = root->next[swd[i]-'a'];
            i++;
        }
        else{
            break;
        }
    }

    if (i == wd.size())
    {
        for (int j = 0; j < root->bwords.size(); j++)
        {
            cout << root->bwords[j] << " ";
        }
        cout<<endl;
        return true;
    }
}

```



```

    }
    return false;
}

int main( )
{
    TrieNode * root = new TrieNode;
    InsertNode(&root, "hehao");
    InsertNode(&root, "ehaoh");
    InsertNode(&root, "haohe");
    InsertNode(&root, "aoheh");
    InsertNode(&root, "facri");
    InsertNode(&root, "et");
    SearchNode(root, "oheha");
    return 0;
}

```

程序输出结果:

hehao ehaoh haohe aoheh

上例中, Trie 树的构建是在预处理阶段完成的, 首先根据字典中的单词来建立字典树, 当建立完字典树后, 查询兄弟单词的效率就会提高很多, 比 Hash 法效率还要高。

Trie 树适用数据量大、重复多, 但是数据种类小可以放入内存的情况。例如, 已知 n (n 很大) 个由小写字母构成的平均长度为 10 的单词, 判断其中是否存在某个字符串是另一个字符串的前缀子串。针对这种问题, 一般可以采用以下 3 种方法。

(1) 迭代法。

对于每一个单词, 都要去查找它前面的单词中是否包含它, 看每个字符串是否为字符串集中某个字符串的前缀, 由于需要不停地进行迭代比较, 所以此时的时间复杂度为 $O(n^2)$ 。

(2) Hash 法。

使用 Hash 方法存储所有字符串的所有前缀子串。而建立存有子串 Hash 的时间复杂度为 $O(n \cdot \text{len})$, 查询的复杂度为 $O(n) \cdot O(1) = O(n)$ 。

(3) Trie 树。

假设要查询的单词是 abcd, 那么在它前面的单词中, 以 b、c、d、f 之类开头的单词则不必考虑, 而只要找以 a 开头的单词中是否存在 abcd 就可以了。同样, 在以 a 开头的单词中, 只要考虑以 b 作为第二个字母的单词即可, 所以建立 Trie 树的复杂度为 $O(n \cdot \text{len})$, 而建立操作与查询操作在 Trie 树中是可以同时执行的。所以, 总的复杂度为 $O(n \cdot \text{len})$, 实际查询的复杂度只是 $O(\text{len})$ 。例如, 有串 911, 911456 输入, 如果要同时执行建立与查询, 过程如下: 首先查询 911, 没有; 然后存入 9、91、911, 再查询 911456, 没有; 然后存入 9114、91145、911456, 而程序没有记忆功能, 并不知道 911 在输入数据中出现过, 所以使用 Hash 必须先存入所有子串, 然后 for 循环查询。而 Trie 树则可以, 存入 911 后, 已经记录 911 为出现的字符串, 在存入 911456 的过程中就能发现而输出答案。反过来也可以, 先存入 911456, 再存入 911 时, 当指针指向最后一个 1 时, 程序会发现这个 1 已经存在, 说明 911 必定是某个字符串的前缀。

8. 堆

堆是一种树形数据结构, 每个结点都有一个值, 而通常所说的堆, 一般是指二叉堆。在堆中, 以大顶堆为例, 堆的根结点的值最大, 且根结点的两个子树也是一个大顶堆, 基于以上特点, 堆适用于海量数据求前 N 大 (用小顶堆) 或者前 N 小 (用大顶堆) 数问题, 其中 N 一般比较小。例如, 当求海量数据前 N 小的数据时, 使用大顶堆, 比较当前元素与大顶堆的最大元素 (即堆顶元素), 如果该元素小于最大元素, 则应该替换该最大元素, 并调整堆的结构

(具体过程见 13.5.7 小节的内容)。当求海量数据前 N 大的数据时, 思路一样。由于采用堆, 只需要扫描一遍即可得到所有的前 n 元素, 所以在海量信息处理中, 效率非常高。

在海量数据处理中, 堆的作用见表 14-2。

表 14-2 堆及其描述

| | 描 述 | 堆 类 型 | 作 用 |
|---|--------------------------|-------|----------|
| 堆 | 求海量数据中前 n 大/小的 值中位数 | 最大堆 | 求前 n 小 |
| | | 最小堆 | 求前 n 大 |
| | | 双堆 | 中位数 |

9. 双层桶法

双层桶不是一种数据结构, 而是一种算法思想, 类似于分治思想。因为元素范围很大, 不能利用直接寻址表, 所以通过多次划分, 逐步确定范围, 最后在一个可以接受的范围内进行。

本文以桶排序进行分析, 桶排序的基本思想是把 $[0, 1)$ 划分为 n 个大小相同的子区间, 每一子区间是一个桶, 然后将 n 个记录分配到各个桶中。因为关键字序列是均匀分布在 $[0, 1)$ 上的, 所以一般不会有多个记录落入同一个桶中。由于同一桶中的记录其关键字不尽相同, 所以必须采用关键字比较的排序方法 (通常用插入排序) 对各个桶进行排序, 然后依次将各非空桶中的记录连接 (收集) 起来即可。这种排序思想的前提是假设输入的 n 个关键字序列随机分布在区间 $[0, 1)$ 之上, 若关键字序列的取值范围不是该区间, 只要其取值均非负, 总能将所有关键字除以某一合适的数, 将关键字映射到该区间上, 但要保证映射后的关键字是均匀分布在 $[0, 1)$ 上的。

桶排序的平均时间复杂度是 $O(n)$, 最坏情况仍有可能是 $O(n^2)$, 一般只适用于关键字取值范围较小的情况, 否则所需桶的数目 m 太多导致浪费存储空间和计算时间。例如, $n=10$, 被排序的记录关键字 k_i 取值范围是 $0 \sim 99$ 之间的整数 (36, 5, 16, 98, 95, 47, 32, 36, 48) 时, 要用 100 个箱子来做一趟排序。

一个桶排序的实例如下:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
    int key;
    node * next;
} KeyNode;

void IncSort(int keys[], int size, int bucketsize)
{
    KeyNode **bucket_table = (KeyNode **) malloc(bucketsize * sizeof(KeyNode *));
    for(int i=0; i<bucketsize; i++)
    {
        bucket_table[i] = (KeyNode *) malloc(sizeof(KeyNode));
        bucket_table[i] -> key = 0; //记录当前桶中的数据量
        bucket_table[i] -> next = NULL;
    }
    for(int j=0; j<size; j++)
    {
        KeyNode *node = (KeyNode *) malloc(sizeof(KeyNode));
        node -> key = keys[j];
        node -> next = NULL;
```

```

        int index=keys[j]/10;
        KeyNode *p=bucket_table[index];
        if(p->key==0)
        {
            bucket_table[index]->next=node;
            (bucket_table[index]->key)++;
        }
        else
        {
            //链表结构的插入排序
            while(p->next!=NULL&& p->next->key<=node->key)
                p=p->next;
            node->next=p->next;
            p->next=node;
            (bucket_table[index]->key)++;
        }
    }
    //打印结果
    for(int b=0;b<bucketsize;b++)
        for(KeyNode *k=bucket_table[b]->next; k!=NULL; k=k->next)
            printf("%d ",k->key);
    printf("\n");
}

int main( )
{
    int array[]={49,38,65,97,76,13,27,49};
    int size=sizeof(array)/sizeof(int);
    IncSort(array,size,10);
    return 0;
}

```

程序输出结果:

13 27 38 49 49 65 76 97

桶排序一般适用于寻找第 k 大的数、寻找中位数、寻找不重复或重复的数字等情况。例如:

(1) 在一个文件中有 10 亿个整数, 乱序排列, 要求找出中位数, 内存限制为 2GB。

(2) 现在有一个 0~30000 的随机数生成器。请根据这个随机数生成器, 设计一个抽奖范围是 0~350000 彩票中奖号码列表, 其中要包含 20000 个中奖号码。

10. MapReduce 法

MapReduce 是云计算的核心技术之一, 是一种简化并行计算的分布式编程模型。它为并行系统的数据处理提供了一个简单、高效的解决方案, 其主要目的是为了大型集群的系统能在大数据集上进行并行工作, 并用于大规模数据的并行运算。

MapReduce 适用于大规模数据集 (通常大于 1TB) 的并行运算, 它的核心操作是 Map 和 Reduce, 即 MapReduce 拆开为 “Map (映射)” 和 “Reduce (化简)”。其中, Map 函数独立地对每个元素进行操作, 它用于把一组键值对映射成一组新的键值对, 即先通过 Map 程序将数据切割成不相关的区块, 分配 (调度) 给大量计算机处理达到分布计算的效果, 然后通过指定并发的 Reduce 函数来将结果汇总, 保证所有映射键值对中的每一个共享相同的键组。

简而言之, 一个映射函数就是对一些独立元素组成的概念上的列表 (如一个测试成绩的列表) 的每一个元素进行指定的操作 (例如, 有人发现所有学生的成绩都被低估了一分, 他可以定义一个 “加 1” 的映射函数, 用来修正这个错误)。而 Map 操作与 Reduce 操作都可以高度并行运行, Map 是把一组数据一对一地映射为另外的一组数据, 其映射的规则由一个函数来

指定。例如，对[1,2,4,8]进行乘2的映射就变为[2,4,8,16]，Reduce是对一组数据进行规约，这个规约的规则是由另外一个函数指定的。例如，对[1,2,4,8]进行求和规约得到的结果是15，而对它进行求积的规约是64。

通过 MapReduce，不会分布式并行编程的程序员也能很容易地将自己的程序运行在分布式系统上。同时，通过该模型，能够充分高效地利用集群中每个机器的资源，适合在集群中处理大规模数据的计算任务，这些优点使得其已经成为云计算平台的主流编程模型。

在架构中，MapReduce API 提供 Map 和 Reduce 处理、GFS 分布式文件系统和 BigTable 分布式数据库提供数据存取。

面对海量数据的处理，分布式的计算方式会导致网络间大量频繁的数据交换，在这种情况下网络带宽相对属于稀缺资源。输入的数据存储在集群中机器的本地磁盘上，这样对有限的带宽来说是有利的。系统按照一个的大小划分数据段，原始文件被划分到各个数据段中。对每个数据段进行备份，分布在不同的机器上。管理机存储这些文件的位置信息，并安排处理这些文件或文件副本的映射任务。如果操作失败，管理机将重新安排映射任务给包含原始文件副本的工作执行。当在集群的工作站运行大型的 MapReduce 操作时，大部分输入数据都可以在本地读取，这样减小了对网络带宽的占用。

海量数据处理的最大难题在于数据规模巨大，使得传统处理方式面临计算能力不足和存储能力不足的瓶颈问题，而基于 Hadoop 可以非常轻松和方便地完成处理海量数据的分布式并行程序，并运行于大规模集群上。

14.3 经典实例分析

有关海量数据处理的-一直以来都是互联网企业笔试面试的重点，此类题目也非常多，但归纳起来，主要有以下3类：top K 问题、重复问题、排序问题。以下将分别对这3类问题进行详细的分析。

14.3.1 top K 问题

在大规模数据处理中，经常会遇到的一类问题：在海量数据中找出出现频率最高的前 K 个数，或者从海量数据中找出最大的前 K 个数，这类问题通常被称为 top K 问题。例如，在搜索引擎中，统计搜索最热门的10个查询词；在歌曲库中统计下载率最高的前10首歌等。

针对 top K 类问题，通常比较好的方案是分治+Trie 树/hash+小顶堆，即先将数据集按照 Hash 方法分解成多个小数据集，然后使用 Trie 树或者 Hash 统计每个小数据集中的 query 词频，之后用小顶堆求出每个数据集中出频率最高的前 K 个数，最后在所有 top K 中求出最终的 top K。

例如：有1亿个浮点数，如何找出其中最大的10000个？

最容易想到的方法是将数据全部排序，然后在排序后的集合中进行查找，最快的排序算法的时间复杂度一般为 $O(n\log n)$ ，如快速排序。而在32位机器上，每个 float 类型占4个字节，1亿个浮点数就要占用400MB的存储空间，对于一些可用内存小于400MB的计算机而言，很显然是不能一次将全部数据读入内存进行排序的。其实即使内存能够满足要求，该方法也并不高效，因为题目的目的是寻找出最大的10000个数即可，而排序却是将所有的元素都排序了，做了很多无用功。

第二种方法为局部淘汰法，该方法与排序方法类似，用一个容器保存前 10000 个数，然后将剩余的所有数字一一与容器内的最小数字相比，如果所有后续的元素都比容器内的 1000 个数还小，那么容器内的这 10000 个数就是最大的 10000 个数。如果某一后续元素比容器内的最小数字大，则删掉容器内最小元素，并将该元素插入容器，最后遍历完这 1 亿个数，得到的结果容器中保存的数即为最终结果了。此时的时间复杂度为 $O(n+m^2)$ ，其中 m 为容器的大小，即 10000。

第三种方法是分治法，将 1 亿个数据分成 100 份，每份 100 万个数据，找出每份数据中最大的 10000 个，最后在剩下的 100×10000 个数据里面找出最大的 10000 个。如果 100 万数据选择足够理想，那么可以过滤掉 1 亿数据里面 99% 的数据。100 万个数据里面查找最大的 10000 个数据的方法如下：用快速排序的方法，将数据分为 2 堆，如果大的那堆个数 N 大于 10000 个，继续对大堆快速排序一次分成 2 堆，如果大堆个数 N 小于 10000，就在小的那堆里面快速排序一次，找第 $10000-n$ 大的数字；递归以上过程，就可以找到第 1w 大的数。参考上面的找出第 1w 大数字，就可以类似的方法找出前 10000 大数字了。此种方法每次需要的内存空间为 $10^6 \times 4 = 4\text{MB}$ ，一共需要 101 次这样的比较。

第四种方法是 Hash 法。如果这 1 亿个数里面有很多重复的数，先通过 Hash 法，把这 1 亿个数字去重复，这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间，然后通过分治法或最小堆法查找最大的 10000 个数。

第五种方法采用最小堆。首先读入前 10000 个数来创建大小为 10000 的小顶堆，建堆的时间复杂度为 $O(m \log m)$ (m 为数组的大小即为 10000)，然后遍历后续的数字，并与堆顶（最小）数字进行比较。如果比最小的数小，则继续读取后续数字；如果比堆顶数字大，则替换堆顶元素并重新调整堆为小顶堆。整个过程直至 1 亿个数全都遍历完为止。然后按照中序遍历的方式输出当前堆中的所有 10000 个数字。该算法的时间复杂度为 $O(nm \log m)$ ，空间复杂度是 10000(常数)。

实际上，最优的解决方案应该是最符合实际设计需求的方案，在实际应用中，可能有足够大的内存，那么直接将数据扔到内存中一次性处理即可，也可能机器有多个核，这样可以采用多线程处理整个数据集。

下面针对不同的应用场景，分析了适合相应应用场景的解决方案。

(1) 单机+单核+足够大内存。

如果需要查找 10 亿个查询词（每个占 8B）中出现频率最高的 10 个，考虑到每个查询词占 8B，则 10 亿个查询词所需的内存大约是 $10^9 \times 8\text{B} = 8\text{GB}$ 内存。如果有这么大的内存，直接在内存中对查询词进行排序，顺序遍历找出 10 个出现频率最大的即可。这种方法简单快速，更加实用。当然，也可以先用 HashMap 求出每个词出现的频率，然后求出频率最大的 10 个词。

(2) 单机+多核+足够大内存。

这时可以直接在内存中使用 Hash 方法将数据划分成 n 个 partition，每个 partition 交给一个线程处理，线程的处理逻辑是同 (1) 类似，最后一个线程将结果归并。

该方法存在一个瓶颈会明显影响效率，即数据倾斜。每个线程的处理速度可能不同，快的线程需要等待慢的线程，最终的处理速度取决于慢的线程。而针对此问题，解决的方法是，将数据划分成 $c \times n$ 个 partition ($c > 1$)，每个线程处理完当前 partition 后主动取下一个 partition 继续处理，直到所有数据处理完毕，最后由一个线程进行归并。

(3) 单机+单核+受限内存。

这种情况下，需要将原数据文件切割成一个一个小文件，如采用 $\text{hash}(x) \% M$ ，将原文件中的数据切割成 M 小文件，如果小文件仍大于内存大小，继续采用 Hash 的方法对数据文件进行

切割,直到每个小文件小于内存大小,这样每个文件可放到内存中处理。采用(1)的方法依次处理每个小文件。

(4) 多机+受限内存。

这种情况下,为了合理利用多台机器的资源,可将数据分发到多台机器上,每台机器采用(3)节中的策略解决本地的数据。可采用 hash+socket 方法进行数据分发。

从实际应用的角度考虑,(1)(2)(3)(4)方案并不可行,因为在大规模数据处理环境下,作业效率并不是首要考虑的问题,算法的扩展性和容错性才是首要考虑的。算法应该具有良好的扩展性,以便数据量进一步加大(随着业务的发展,数据量加大是必然的)时,在不修改算法框架的前提下,可达到近似的线性比;算法应该具有容错性,即当前某个文件处理失败后,能自动将其交给另外一个线程继续处理,而不是从头开始处理。

top K 问题很适合采用 MapReduce 框架解决,用户只需编写一个 Map 函数和两个 Reduce 函数,然后提交到 Hadoop (采用 Mapchain 和 Reducechain) 上即可解决该问题。具体而言,就是首先根据数据值或者把数据 hash(MD5)后的值按照范围划分到不同的机器上,最好可以让数据划分后一次读入内存,这样不同的机器负责处理不同的数值范围,实际上就是 Map。得到结果后,各个机器只需拿出各自出现次数最多的前 N 个数据,然后汇总,选出所有的数据中出现次数最多的前 N 个数据,这实际上就是 Reduce 过程。对于 Map 函数,采用 Hash 算法,将 Hash 值相同的数据交给同一个 Reduce task;对于第一个 Reduce 函数,采用 HashMap 统计出每个词出现的频率,对于第二个 Reduce 函数,统计所有 Reduce task,输出数据中的 top K 即可。

直接将数据均分到不同的机器上进行处理是无法得到正确的结果的。因为一个数据可能被均分到不同的机器上,而另一个则可能完全聚集到一个机器上,同时还可能存在具有相同数目的数据。

Top K 问题还有很多应用场景,尤其是在程序员面试笔试中有很多实例,它们都可以采用上述方法解决。以下是一些历年来经常被各大互联网公司提及的该类问题。

(1) 有 10000000 个记录,这些查询串的重复度比较高,如果除去重复后,不超过 3000000 个。一个查询串的重复度越高,说明查询它的用户越多,也就是越热门。请统计最热门的 10 个查询串,要求使用的内存不能超过 1GB。

(2) 有 10 个文件,每个文件 1GB,每个文件的每一行存放的都是用户的 query,每个文件的 query 都可能重复。按照 query 的频度排序。

(3) 有一个 1GB 大小的文件,里面的每一行是一个词,词的大小不超过 16 个字节,内存限制大小是 1MB。返回频数最高的 100 个词。

(4) 提取某日访问网站次数最多的那个 IP。

(5) 10 亿个整数找出重复次数最多的 100 个整数。

(6) 搜索的输入信息是一个字符串,统计 300 万条输入信息中最热门的前 10 条,每次输入的一个字符串为不超过 255B,内存使用只有 1GB。

(7) 有 1000 万个身份证号以及他们对应的数据,身份证号可能重复,找出出现次数最多的身份证号。

14.3.2 重复问题

在海量数据中查找出重复出现的元素或者去除重复出现的元素也是常考的问题。针对此类问题,一般可以通过位图法实现。例如,已知某个文件内包含一些电话号码,每个号码为 8 位

数字，统计不同号码的个数。

本题最好的解决方法是通过使用位图法来实现。8 位整数可以表示的最大十进制数值为 99999999。如果每个数字对应于位图中一个 bit 位，那么存储 8 位整数大约需要 99MB。因为 1B=8bit，所以 99Mbit 折合成内存为 $99/8=12.375\text{MB}$ 的内存，即可以只用 12.375MB 的内存表示所有的 8 位数电话号码的内容。

程序示例如下：

```
#include <iostream>
#include <time.h>
using namespace std;

#define BITWORD 32
#define ARRNUM 100
int mmin = 10000000;
int mmax = 99999999;
int N = (mmax-mmin+1);
#define BITS_PER_WORD 32
#define WORD_OFFSET(b) ((b) / BITS_PER_WORD)
#define BIT_OFFSET(b) ((b) % BITS_PER_WORD)

void SetBit(int *words, int n)
{
    n -= mmin;
    words[WORD_OFFSET(n)] |= (1 << BIT_OFFSET(n));
}

void ClearBit(int *words, int n)
{
    words[WORD_OFFSET(n)] &= ~(1 << BIT_OFFSET(n));
}

int GetBit(int *words, int n)
{
    int bit = words[WORD_OFFSET(n)] & (1 << BIT_OFFSET(n));
    return bit != 0;
}

int main( )
{
    int i;
    int j;
    int arr[ARRNUM];
    int* words = new int[1 + N/BITS_PER_WORD];
    if(words == NULL)
    {
        cout << "new error\n" << endl;
        exit(0);
    }

    int count = 0;
    for (i = 0; i < N; i++)
    {
        ClearBit(words, i);
    }

    srand( (unsigned)time( NULL ) );
```

```

printf("数组大小:%d\n", ARRNUM);

for (j = 0; j < ARRNUM; j++)
{
    arr[j] = rand() % N;
    arr[j] += mmin;
    printf("%d\t", arr[j]);
}

for (j = 0; j < ARRNUM; j++)
{
    SetBit(words, arr[j]);
}
printf("排序后 a 为:\n");
for (i = 0; i < N; i++)
{
    if (GetBit(words, i))
    {
        printf("%d\t", i+mmin);
        count++;
    }
}
printf("总个数为:%d\n", count);
delete[] words;
words = NULL;
return 0;
}

```

上例中，采用时间作为种子，产生了 100 个随机数，对这 100 个数进行位图法排序，进而找出其中重复的数据。与此问题相似的面试笔试题还有：

- (1) 10 亿个正整数，只有 1 个数重复出现过，要求在 $O(n)$ 的时间里找出这个数。
- (2) 给定 a、b 两个文件，各存放 50 亿个 url，每个 url 各占用 64B，要求在 $O(n)$ 的时间里找出 a、b 文件共同的 url。
- (3) 给 40 亿个不重复的 unsigned int 的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那 40 亿个数当中？

14.3.3 排序问题

海量数据处理中一类常见的问题就是排序问题，即对海量数据中的数据进行排序。例如，一个文件中有 9 亿条不重复的 9 位整数，对这个文件中的数字进行排序。

针对这个问题，最容易想到的方法是将所有数据导入到内存中，然后使用常规的排序方法，如插入排序、快速排序、归并排序等各种排序方法对数据进行排序，最后将排序好的数据存入文件。但这些方法却不能在此适用，由于数据量巨大，在 32 位机器中，一个整数占用 4 个字节，而 9 亿条数据共占用 $9 \times 10^8 \times 4B$ ，大约需要占用 3.6GB 内存，对于 32 位机器而言，很难将这么多数据一次载入到内存，更不谈进行排序了，所以此种方法一般不可行，需要考虑其他方法。

方法一：数据库排序法。将文本文件导入到数据库中，让数据库进行索引排序操作后提取数据到文件。该方法虽然操作简单、方便，但是运算速度较慢，而且对数据库设备要求比较高。

方法二：分治法。通过 hash 将 9 亿条数据分为 20 段，每段大约 5000 万条，大约占用 $5 \times 10^6 \times 4B = 200MB$ 空间，在文件中依次搜索 0~5000 万，50000001~1 亿……将排序的结果存

入文件。该方法要装满 9 位整数，一共需要 20 次，所以一共要进行 20 次排序，需要对文件进行 20 次读操作。该方法虽然缩小了每次使用的内存空间大小，但是编码复杂，速度也慢。

方案三：位图法。考虑到最大的 9 位整数为 999999999，由于 9 亿条数据是不重复的，可以把这些数据组成一个队列或数组，让它有 0~999999999（一共 10 亿个数）个元素数组下标表示数值，结点中用 0 表示没有这个数，1 表示存在这个数，判断 0 或 1 只用一个 bit 存储就够了，而声明一个可以包含 9 位整数的 bit 数组，一共需要 10 亿/8，大约 120MB 内存，把内存中的数据全部初始化为 0，读取文件中的数据，并将数据放入内存。比如读到一个数据为 341245909，那就先在内存中找到 341245909 这个 bit，并将 bit 值置为 1，遍历整个 bit 数组，将 bit 为 1 的数组下标存入文件，最终得到排序后的内容。

此类排序问题的求解方法一般都是采用上述方法。海量数据处理中与此类似的问题还有以下几种：

（1）一年的全国高考考生人数为 500 万，分数使用标准分，最低 100 分，最高 900 分，不存在成绩为小数的情况，把这 500 万考生的分数排序。

（2）一个包含 n 个正整数的文件，每个正整数小于 n ， n 等于 1000 万，并且文件内的正整数没有重复和关联数据，输出整数的升序排列。

致 谢

本书所涉及的内容，有一部分原始资料来源于网络，有些思想也是受到网络上大量博主、分享者的启发，他们是本书背后默默无闻的英雄。但由于有些地方无法获悉原作者信息，联系不上原作者，所以无法在此对这些人的名字予以公布，特在此处对这些人表示衷心的感谢，并致以最崇高的敬意。